Closing the Gap between XML and Relational Database Technologies: State-of-the-Practice, State-of-the-Art and Future Directions

Mary Ann Malloy
The MITRE Corporation, USA

Irena Mlynkova
Charles University, Czech Republic

ABSTRACT

As XML technologies have become a standard for data representation, it is inevitable to propose and implement efficient techniques for managing XML data. A natural alternative is to exploit tools and functions offered by relational database systems. Unfortunately, this approach has many detractors, especially due to inefficiency caused by structural differences between XML data and relations. But, on the other hand, relational databases represent a mature, verified and reliable technology for managing any kind of data including XML documents. In this chapter, we provide an overview and classification of existing approaches to XML data management in relational databases. We view the problem from both state-of-the-practice and state-of-the-art perspectives. We describe the current best known solutions, their advantages and disadvantages. Finally, we discuss some open issues and their possible solutions.

1. INTRODUCTION

Without a doubt, the eXtensible Markup Language (XML) (Bray et al., 2006) is one of the most popular contemporary formats for data representation. It is well-defined, easy-to-use and involves various recommendations such as languages for structural specification, transformation, querying, updating, etc. This wide popularity naturally has evoked intense effort to propose faster and more efficient methods and tools for managing and processing XML data. Soon it became possible to distinguish several different directions. The four most popular approaches are: methods that store XML data in a classical file system; methods that store and process XML data using a relational database system; methods that exploit a pure object-oriented approach; and, native XML methods that use special indices, numbering schemes and/or data structures particularly suitable for the
tree structure of XML data. Naturally, each of these approaches has both keen advocates and detractors who emphasize its particular advantages or disadvantages.

The situation is not good especially for file system-based and pure object-oriented methods. The former approach suffers from an inability to query without any additional pre-processing of the data; whereas the latter approach fails in particular in finding a corresponding efficient and comprehensive implementation. As expected, the highest-performance techniques are the native ones, since they are tailored particularly for XML processing and do not need to artificially adapt existing structures to a new purpose. Nevertheless, the most practically used methods exploit features of relational databases. Although researchers have already proven that native XML strategies perform much better, they still lack one important aspect: a robust implementation verified by years of both theoretical and practical effort.

If we consider this problem from an alternative viewpoint, we realize that considerable amounts of data in practical use are still stored in relational databases. Legacy relational stores are well-established and reliable enough that their existence is entrenched and they are unlikely to disappear anytime soon (Bruce, 2007). Developers must sustain existing investments in applications predicated on a relational architecture while, at the same time, adapting them to the heterogeneous and message-driven nature of XML. A typical use case may involve mapping web document content from an XML representation into a relational database. Not only does this help insulate naïve web clients from the underlying and perhaps less familiar XML technologies, it also positions the information for storage and query via the more mature technologies associated with RDBMSs. Alternatively, middleware may permit XML sophisticates to view and query relational contents as though they were XML documents, and vice versa. For the foreseeable future, some hybrid of these solutions is likely to be developed, although the major relational database vendors are already providing embedded XML support.

Consequently, currently there are many efforts focused on database-centric XML data management. The researchers focus on more efficient strategies to query evaluation, database vendors more and more support XML and even the SQL standard has been extended by SQL/XML which introduces a new XML data type and operations for XML data manipulation. But, although the amount of existing solutions is large, there are still unsolved problems, open issues and aspects to be improved.

1.1. Goals of the Chapter

The main aim of the chapter is to study the problem of XML and relational technology convergence from various points of view. Firstly we view the problem fairly generically, in terms of the various data sources, integration approaches, processing models and types of applications to be considered. Secondly, we review commercial off-the-shelf (COTS) offerings that might affect a business’ strategic choices between XML and relational processing and representations, and what the major database vendors are doing.

Thirdly, we study the current trends in the research area. The main concern of these techniques is the choice of how XML data are stored into relations, so-called XML-to-relational mapping or decomposition to relations. We provide an overview of existing approaches of XML-to-relational mapping strategies and their (dis-)advantages especially with respect to the efficiency of query evaluation. We focus particularly on the most efficient and promising representatives at present: adaptive methods. Since adaptability is the ability to adapt the target relational schema to
information and/or requirements which specify the future application, such methods have better performance results than the general methods designed to be reasonably efficient in all cases.

An important aim of the chapter is to identify the most striking related open issues and unsolved state-of-the-practice and state-of-the-art problems. From a practical viewpoint, we will describe areas where richer methodologies are still needed to support the coordinated exploitation of XML and RDBMS technologies. In addition, this discussion will suggest that additional engineering efforts must be invested in order to achieve acceptable results across impacted information sharing dimensions, such as semantic, physical, size and homogeneity characteristics. Keeping these criteria in the forefront will ensure that advances on the database front do not cripple the highly touted “quick-and-easy” appeal that led developers down the XML road in the first place. From a research viewpoint, the open problems focus on issues and challenges in XML-to-relational mapping strategies. In particular, we show that although the adaptive strategies seem to be the most promising candidates, we can go even further and adapt the schema dynamically.

1.2. Roadmap

The rest of the chapter is structured as follows. First we provide some general background material to overview the problems and tradeoffs associate with mingling XML and relation data. The next section analyzes the existing approaches to XML management in relational databases from a state-of-the-practice perspective and the following one from a state-of-the-art perspective. We then describe and discuss open issues and possible solutions and, finally, offer conclusions.

2. GENERAL BACKGROUND

In this section, we briefly summarize the various data sources, integration approaches, processing models and applications types that comprise the problem space of XML and relational data management. This lays the foundation for the remainder of the chapter.

We also note that any meaningful discussion of XML and relational technology convergence must be grounded in some foundational topical knowledge. To fully appreciate this chapter, readers are assumed to possess a reading familiarity with the following technologies: relational databases (Codd, 1970; Date, 1999); SQL (Hare, 2007); Extensible Markup Language (XML) (Bray, 2006); XML Schema (Biron, 2004; Thompson, 2004); XQuery and XPath (Fernández, 2007); SQL/XML (Joint Technical Committee ISO/IEC JTC1, 2003); XSLT (Kay, 2007); and XQuery Update Facility (Robie, 2007). Seminal sources have been cited in the References for those readers who require more background information.

2.1. Data Sources

The data sources feeding information processing applications are many and varied. In the context of this discussion, consideration is limited to legacy relational content and XML source documents. In the former case, assume the data is stored in the tables and columns of a relational database, created and maintained by some legacy applications. Users may wish to continue processing this information in the legacy fashion, viewing and manipulating it as columnar values
within tables via relational-aware applications. Or, they may wish to be provided with a virtual XML view of the relational data, presumably for accessing, querying and updating it via XML-aware applications.

There is variance in how XML is being used in industry; therefore XML source data comes in several flavours. One useful binning is to consider the degree of “structure” – or “data-centricity”, versus “document-centricity” – the sources exhibit (Bourret, 2005). Semi-structured XML may be created for human consumption (e.g., web pages, office documents). Such document-centric sources can be the realm of web publishers or back-office residents who use content management systems that enable storing, updating and retrieving XML documents in a shared repository. Typically these systems include editors and engines that provide some or all of the following: versioning, revisions and access controls; mapping to alternative formats; collaboration; web publishing; indexing and search. This category may also include documents that contain both freeform text and marked up content.

The other primary usage of XML is highly structured and data-centric. Here XML is used as a storage or interchange format for data that is regularly ordered and targeted for machine-processing as opposed to human-readability (e.g., SOAP messages). These documents may or may not be persisted in whole or in part, depending on their specific employments.

2.2. Integration Design Patterns

Many applications require that XML data – irrespective of its origin – be persisted in a database that allows for more sophisticated storage and retrieval. There are two fundamental approaches for accomplishing this in a relational context: CLOB (Character Large Object) and decomposition to relations (Rys, 2005).

CLOB (Character Large Objects)

This straightforward scheme stores XML data as unstructured content, typically placing each document in its entirety into a single column in a relational table. In this way, the textual fidelity of the document and its internal relationships are preserved. A given application might store multiple types of documents in different columns or even in the same column. Supplementary information (e.g., unique identifiers, date-time stamps) might be added in other tables or columns to support document management. A disadvantage is that treating the document as “just a text file” can compromise query and search algorithms, since they cannot take advantage of structural and semantic content exposed through XML markup. In addition, it is typical that entire documents must be read into memory one after another, parsed and searched, which poses an intimidating challenge in terms of processing overhead. This effect can be moderately improved if indices are built at insert time; yet extraction of document fragments will continue to incur expensive parsing.

Still, CLOB can be a reasonable representation choice in a number of situations; among them:

- The typical extraction is a full document and its performance is of high priority.
- Documents need to be kept intact (e.g., legal, regulatory, business requirements).
- Few attribute/element updates are expected within documents.
- Mapping the document structure into a relational schema is too complicated.
- Documents with different or evolving schemas need to be managed together.
- Documents are typically searched or identified based on “side” or “property” tables, not contents (Lapis, 2005).

Decomposition to Relations

In this approach, XML content and associated metadata are meaningfully distributed (or decomposed) across the columns of one or more relational tables. Issues and choices arise from the fact that there must be some mechanism that determines the corresponding set of tables used for this representation. The mappings can be non-trivial, and this topic continues to be discussed in the literature. Where the original input data was in XML form, the term “shredding” is used and the mapping preserves both content and structural relationships so that it should be possible to recover the original XML document in the future.

It should be noted that it can be quite difficult to define the SQL storage structure for complex XML schemas, with the attendant overhead of parsing and shredding the data for insertion and pulling them back together whenever they are retrieved. Shredding into automatically generated tables works well for fairly simple flat XML schemas, but rapidly becomes unmanageable when applying complex nested XML schemas as allowed by the standard with all the attendant overhead challenges one might expect. The State-of-the-Art section of this chapter delves into this topic in greater detail.

Decomposition is a reasonable choice when one or more of the following contingencies are met:

- The source document structure is no longer relevant once the data has been posted.
- Document content must be integrated with existing relational data.
- Document structure yields a reasonable mapping to a relational schema.
- XML source schema(s) changes are unforeseen or rare.
- Existing relational applications must be used.
- The structure of extracted XML documents is different from that of input sources.
- Anticipated queries are expressible in SQL and/or the mapping to XML-aware query language is easy or not needed.
- Few individual attribute/element updates are expected; update performance is of high importance (Lapis, 2005).

Furthermore, CLOB and decomposition can be co-mingled: part of the XML document might be shredded, while part might be maintained contiguously in a single column; or certain elements of the CLOB might be duplicated as “shredded out” columns. Relational databases that have been extended to add XML functionality may implement a native XML type. Because an automated “one-size-fits-all” transformation of arbitrary documents into the type is unlikely to be satisfactory in many contexts, hand-crafted optimization may be desirable or even required, and this must be repeated for each new document type or at any time that an existing document type changes.

If an XML-native database is available, XML source documents may be stored there, apart from any relational data. It would then be possible to use XML-aware applications to process the XML stores either separately from the relational data or in an integrated fashion. XML storage may be elected in the following circumstances:
• High performance is required whenever processing XML content.
• XML schemas are too complex, numerous, or variable to be mapped to a relational schema.
• The internal structure of the source documents must be preserved.
• The queries required are not easily expressible in SQL.
• The structure of extracted XML documents is different from that of input sources (Lapis, 2005).

A tighter integration might migrate or map the legacy relational data into the XML stores or views if practicable. These sorts of processing and application considerations are discussed in the next two sections.

2.3. Processing Models

When viewed from a high level, there are but three processing models for supporting XML-aware applications on top of a database. XML functionality may be embedded in a custom application. For example, “canned” XML-aware queries (expressed, e.g., in XPath or XQuery language) can be mapped a priori into SQL equivalents which are populated via a template that sits on top of a relational database. A middleware component may be used to provide an end application with a virtual XML view of a relational database (or vice versa). In this case, the middleware acts as a mediator, translating data and/or queries into the representations needed to interact with the user or application on the one hand, and the database on the other. Or, XML processing can be supported via functions or procedures as extensions of the database itself, logically co-mingling them (Draper, 2003).

2.4. Application Types

Application types (i.e., what is to be done with the data) inform the choice of processing models and which integration approaches are better than others for converging XML and relational technologies. These types also constitute a “trade space” where research and optimizations are likely to occur: extraction, querying and updating (Draper, 2003).

The fundamental operation is extraction. The most basic of such applications extracts whole XML documents from a repository. In this case, it is presumed that a document would be stored contiguously in its entirety, or its constituent parts appropriately linked so that the original document could be recomposed upon extraction. Extraction of fragmentary documents additionally requires selection: the ability to access individual document elements, attributes and values and to evaluate predicate conditions to retrieve the desired subset. A naïve way of accomplishing this is to extract entire XML documents from the repository, and then evaluate the selection conditions as a post-processing step; but a more efficient approach is to leverage the database to perform those operations prior to – or as an integral part of – the extraction step.

The querying operation requires XML data navigation, selection, transformation, sorting and aggregation, and it may draw on multiple documents to create a single output. Querying is in fact a generalization of the “transformation” concept. The defining characteristic of this application type is the ability to select and combine individual pieces of data in complex ways, creating new
XML constructs that may not even have existed in the source data. Querying can be accomplished as a post-processing step as discussed above; but again it is desirable to have the database do the work, either by mapping the user’s query into its native language or by using an extended XML-aware query language, depending on the underlying database representation.

Finally, updating operations require the ability to uniquely identify nodes that need to be changed. The difference between the original document and its updated version must be translated into an appropriate set of updates against the database. In a document repository setting, the user might simply replace the entire document with the updated version! Alternatively, the application must be provided with a means of addressing selected nodes, either directly or via an application programming interface (API), to effect the changes. How complex this becomes depends on how the data were mapped into the database in the first place, as discussed earlier. Another concern is the need to coordinate with the underlying database engine regarding transactions and locking during the physical update process.

Still another perspective is accessing persisted XML documents through traditional SQL interfaces to avoid the costly conversion of legacy applications to an XML-aware query language. In this case, middleware might be leveraged to do the mediation.

3. STATE-OF-THE-PRACTICE PERSPECTIVES

The emergence of XML and e-commerce supported by the internet has created new opportunities for businesses to exchange information in ways previously possible only through narrowly defined interchange formats. So, despite the dominant position of relational database management systems (RDBMSs), the growth of XML is forcing businesses to explore ways to function seamlessly in both XML and traditional relational situations. Despite the dominant position of RDBMSs, from an operational viewpoint, the growth of XML is forcing businesses to explore ways to function seamlessly in both XML and traditional relational situations.

With XML becoming the de facto standard for exchanging and querying information over the web, new challenges have been created for persistent information storage and processing. Most contemporary enterprises need to support both XML and relational data for ease of information exchange and increased flexibility. This part of the chapter examines some ways practitioners are processing XML and relational data in parallel, co-mingling them, and migrating content to shared representations and processing models. Our discussion assumes the enterprise that is managing data requires the following:

- to sustain or increase use of XML as a data format where appropriate;
- to administer and protect XML data;
- to archive XML data for business requirements and/or legal obligations;
- to employ existing relational applications with XML data where appropriate; and
- to integrate XML and relational data where appropriate.

Some of the typical scenarios that are motivating the need for the convergence of these technologies are: managing, querying and transforming document content for day-to-day operations as well as web sites; integrating diverse document content from heterogeneous systems by converting them into more flexibly formatted XML documents; to represent semi-structured
data, because change is occurring ever more rapidly, and XML helps mitigate some related forward- and backward compatibility problems. (Bourret, 2005) discusses many specific use cases on his website.

3.1. State-of-the-Practice Solutions

Users may need to extract data from a database in XML format, store XML data into a database, or both. Middleware-based solutions are available when there is a need to retrieve content from databases and return it as XML and vice versa; but there is quite a bit of variability in the functionalities they provide, whether default mappings are performed between the documents and tables, and to what degree user intervention is permitted or required to define the transformations. We discuss two popular solutions first: SQL/XML and XQuery.

SQL/XML: A Relational-facing Solution

At one time there was no standard way to access XML data from relational databases. This changed with the development of the SQL/XML standard by the SQLX group (Eisenberg, 2002; Krishnaprasad, 2005). SQL/XML is an International Organization for Standardization (ISO) effort that integrates SQL and XML. It specifies SQL-based extensions for using XML in conjunction with SQL. It includes a definition of a native XML data type, implicit and explicit ways of generating XML from relational data, an implicit default mapping from relational data to XML, and a set of XML-generating operators that can be used to create a fixed hierarchy populated directly from a set of relational values.

The collective effect allows XML to be manipulated by SQL and relational content by an XML-aware query-language like XQuery (discussed next) once they have been appropriately rendered via the built-in mappings. It also works well in traditional SQL environments. These functions allow users to continue to think of their tasks in terms of SQL if they choose to do so, and to construct new XML elements or attributes using values from various sources including, but not limited to, relational tables. For example, SQL/XML allows the user to embed XQuery expressions in SQL expressions and return output values of type XML or as a relational table.

XQuery: An XML-facing Solution

Middleware products have been delivering Java components or C++ programs that provide developers with options for presenting and exchanging their relational data as XML and for processing relational and XML data together. Developers can potentially replace that with a few lines of code involving XQuery. XQuery is a standardized language for combining documents, databases, web pages and almost anything else. As stated in the World Wide Web Consortium’s W3C XQuery specification, “a query language that uses the structure of XML intelligently can express queries across all these kinds of data, whether physically stored in XML or viewed as XML via middleware…XQuery…is designed to be broadly applicable across many types of XML data sources” (Fernández, 2007).

XQuery offers SQL-like functionalities to directly manipulate web documents in the ways developers and information producers/consumers have come to expect from their RDBMS experiences. The W3C specification provides this native XML query language as a target for integration platforms and components as a replacement for proprietary middleware languages and
web application development languages. Although it was initially conceived as a query language for large collections of XML documents, it can also be used to transform them.

The XQuery language is based on a tree structure model of XML document content consisting of seven kinds of nodes: documents, elements, attributes, text, comments, processing instructions and namespaces. It uses XPath expression syntax to address specific parts of an XML document. It should be noted that this may be either a real physical document or a virtual view of database content as an XML document. (N.B., XPath is a language for selecting nodes from an XML document and for computing string, numeric or boolean values from content.) The XPath expression syntax is supplemented with the “FLWOR” (pronounced “flower”) expressions FOR, LET, WHERE, ORDER BY, and RETURN, as well as syntax for constructing new XML documents using an XML-like syntax or dynamic node constructors. It is in this sense that it is a programming language that can express arbitrary XML to XML data transformations.

Although XQuery currently lacks features for full search and updating XML documents or databases, commercial RDBMSs are embedding XQuery support as a means to expose relational data as an XML data source. RDBMSs without this native XQuery support can still delegate this responsibility to middleware. In addition, as of August 2007, the W3C’s XQuery Update Facility progressed to last call working draft status (Robie, 2007). This document defines an update facility that extends XQuery with expressions that can be used to make persistent document changes.

3.2. Vendor Solutions

In addition to middleware offerings, most popular databases are now XML-enabled. That is, they have native support for converting relational data to XML and back. In fact, every major relational database vendor has proprietary extensions for using XML with its product, but each takes a completely different approach, and there is little interoperability among them. The “Big Three” vendors (i.e., IBM, Oracle and Microsoft) are moving towards second or third generation support, which means full XML support, preserving the whole XML document; and they provide some form of XQuery support over the data. This landscape is still changing, and increasingly mature product versions are emerging with ever-improving features. The next few paragraphs provide a snapshot of current capabilities.

IBM DB2

IBM provides a truly unified XML/relational database, supporting the XML data model from the client through the database, “down to the disk and back again” through a first-class XML data type. By deeply implementing XML into a database engine that previously was purely relational, IBM offers superior flexibility and performance relative to other offerings. A high-level view of DB2 with native XML support is shown in Figure 1.

Figure 1. IBM DB2 XML Support
DB2 manages both conventional relational and XML data. As depicted in the Storage component of the figure, relational and XML data are stored in different formats that match their respective models: relational as traditional row-column structures; and XML as hierarchical node structures. Both types of storage are accessed via the DB2 engine which processes plain SQL, SQL/XML and XQuery in an integrated fashion.

SQL and XQuery are handled in a single modelling framework, avoiding the need to translate queries between them, via so-called bilingual queries that give developers the flexibility to use the language that matches not just application needs but also their skills. Applications can continue to use SQL to manipulate relational data or the XML store. SQL/XML extensions enable publishing relational data in XML format based on data retrieved by embedding XPath or XQuery into SQL statements. XML applications typically use XQuery to access the XML store; yet XQuery queries can optionally contain SQL to combine and correlate XML with relational data. Further details can be found in (Nicola, 2005; IBM, 2005).

Oracle XML DB

Oracle has been steadily evolving its support for XML since 1998, moving toward flexible, high-performance, scalable XML storage and processing. With new version releases every few years, they have progressed from loosely-coupled XML APIs, to XML storage and repository support, later adding XQuery then binary XML storage and indexing. Figure 2 provides a pictorial overview of XML DB’s features.

Figure 2. Oracle XML DB Features
XML DB implements the major W3C standards (e.g., XML, Namespace, XPath, XML Schema, XSLT). They claim the first major implementation of XQuery as well as support for SQL/XML. This hybrid database provides SQL-centric access to XML content, and XML-centric access to relational content. Multiple XML storage options allow tuning for optimal application performance. An XML DB repository is a nice addition for serving document-centric needs. Further details can be found in (Drake, 2007).

Microsoft SQL Server

A significant portion of Microsoft’s SQL Server architecture is shown in Figure 3. This product features XML storage, indexing and query processing. The XML data type provides a simple mechanism for storing XML data by inserting it into an untyped XML column. The XML data type preserves document order and is useful for applications such as document management applications. Alternatively, XML Schemas may be used to define typed XML; this helps the database engine to optimize storage and query processing in addition to providing data validation. The SQL Server can also handle recursive XML Schemas as well as server-side XQuery.

Microsoft still marches to its own drummer in some respects. Their SQLXML mapping technology is used to layer an XML-centric programming model over relational data stored in tables at the server. (Note SQLXML is completely different from SQL/XML; the similarity in names can cause quite a bit of confusion.) The mapping is based on defining an XML schema as an XML view. This provides a bi-directional mapping of an XML Schema to relational tables. This approach can be used for bulk loading XML data into tables and for querying the tables. Document order is not preserved, however, so the mapping technology is useful for XML data processing as opposed to XML document processing. Microsoft still advocates sticking with a relational model for structured data with a known schema. Further information can be found in (Pal, 2005).

*Figure 3. Microsoft SQL Server Architecture*
4. STATE-OF-THE-ART PERSPECTIVES

If we consider the problem of XML data management in RDBMSs from the research perspective, we find out that the main concern of the techniques is to find the most efficient XML-to-relational mapping, i.e., shredding the data to relations so that the respective queries can be evaluated most efficiently. An XML query (expressed in XPath or XQuery) posed over the data stored in the database is then translated to a set of SQL queries (which is usually a singleton) and the resulting set of tuples is transformed to an XML document. We speak about reconstruction of XML fragments. The way the query can be translated is usually directly determined by the respective mapping strategy, so the key problem is to find the optimal mapping.

Based on whether they exploit or omit information from the XML schema, we can distinguish so-called generic (or schema-oblivious) (e.g., Florescu et al., 1999) and schema-driven (e.g., Shanmugasundaram et al., 1999; Mlynkova et al., 2004) methods. From the point of view of the input data we can distinguish so-called fixed methods (e.g., Florescu et al., 1999; Shanmugasundaram et al., 1999; Mlynkova et al., 2004) which store the data purely on the basis of their model, and adaptive methods (Ramanath et al., 2003; Klettke et al., 2000), where sample XML documents and XML queries also are taken into account. In addition, there are techniques based on user involvement. These can be divided into user-defined (Amer-Yahia, 2003) and user-driven (Amer-Yahia et al., 2004; Balmin et al., 2005). In the former case, a user is expected to define both the relational schema and the required mapping; whereas in the latter case, a user specifies just local changes to a default mapping.

But although the number of existing works is enormous, there is probably no universally efficient strategy. Each of the existing approaches is suitable for selected applications; at the same time, there can usually be found cases where it can be highly inefficient. In general, the development of XML-to-relational mapping strategies tends to exploit more and more information about the target application. It seems to be the most promising approach, since
information on the future usage of the stored data can have such an impact on the efficiency of query evaluation. For better clarity, we provide several illustrative situations.

In the first simple motivating example, consider the following DTD fragment (assuming that the undeclared elements have #PCDATA data type):

```xml
<!ELEMENT employee (name, address)>
<!ELEMENT name (first, middle?, last)>
<!ELEMENT address (city, country, zip)>
```

the natural target relational schema (for simplicity omitting data types and obvious keys and foreign keys) would involve the relation:

```sql
employee1(name_first, name_middle, name_last, address_city, address_country, address_zip)
```

Suppose we know that the user always retrieves an employee's name as a whole. In other words, typical queries over the data involve queries of the form:

```sql
//employee/name
```

but not of the form:

```sql
//employee/name/first
//employee/name/middle
//employee/name/last
```

Then a more efficient relation would be:

```sql
employee2(name, address_city, address_country, address_zip)
```

Another classical example involves updatability of the data, which can be a crucial deciding feature of database storage strategies. On the one hand, we could know that the data will not be updated too much or at all, but we need an efficient query evaluation. On the other hand, there could be a strong demand for efficient data updates; in which case, queries are of marginal importance. And there are of course cases which require efficient processing of both. Naturally, the appropriate storage strategies differ considerably. Where efficient query processing is required, various indices and numbering schemes can be exploited to speed up query evaluation, but at the cost of correspondingly expensive updates. Efficient updates, conversely, require the simplest information about mutual data relationships. And if both efficient queries and updates are required, compromise is unavoidable. And such decisions can be made correctly only if we have appropriate information.

As the last example, let us consider the question of data redundancy. Without any additional information, the optimal storage strategy is the so-called fourth normal form (4NF) schema decomposition into relations (Balmin et al., 2005) which can be achieved, for example, using the classical Hybrid algorithm (Shanmugasundaram et al., 1999), one of the fixed mapping methods. The decomposition does not involve data redundancy or violation of any normal form; i.e., it results in a database schema with the smallest number of relations and null attributes. But, similar
to database design, there can be reasonable real-world cases when the data should not strictly follow the rules of normal forms and their moderation can lead to more efficient query processing.

Obviously, in general, there can hardly exist a universally efficient mapping strategy suitable for all future applications. There are reasonably efficient methods which perform well in typical cases, but we can always find a real-world application which requires different treatment. Hence, the natural solution is to exploit the idea of adaptability and automatically adapt the target schema to the current requirements.

4.1. A Look into History

If we want to exploit the idea of adaptability, we need to acquire some additional information according to which the target relational schema can be adapted. The mapping strategy in the first approaches to XML-to-relational mapping – i.e., the fixed methods – was based purely on the data model of XML documents. The methods were able to store any kind of XML data since they viewed XML documents as general labelled trees. But, the efficiency of query evaluation was quite low due to numerous join operations, and increases in efficiency were gained at the cost of increase of space overhead.

Next, researchers came up with a natural idea to exploit structural information extracted from XML schemas. The so-called schema-driven methods were based on the idea that the structure of the target relational schema is created according the structure of the source XML schema. Assuming that a user specifies the XML schema as precisely as possible with respect to the related data, we will also get a more precise relational XML schema. The problem is that DTDs are usually too general. For example, recursion or the “*” operator can in general allow infinitely deep or wide XML documents. In both cases, the respective XML documents are much simpler (Mlynkova et al., 2006) and, thus, the effort spent on processing all the complex schema constructs is wasted.

This problem was somewhat mitigated by exploring ways by which the schema-driven approaches could leverage information extracted from XSDs (XML Schema definitions) (Mlynkova et al., 2004). The XML Schema language enables the structure of XML data to be described more precisely, especially in the case of data types. But, on the other hand, although its expressive power is higher, most of these new constructs can be considered “syntactic sugar”. Hence, exploitation of XML Schema constructs does not significantly impact on XML processing efficiency and it is necessary to exploit further information.

An alternative approach to improving the fixed mapping strategies led to methods that focus on the preservation of constraints, such as key and foreign key constraints, functional dependencies or semantic constraints, specified in the source XML schema in the target relational schema (e.g., Chen et al., 2003; Davidson et al., 2007). Such approaches reduce redundancy and improve efficiency for selected queries and, especially, update operations. But, in general, the methods suffer from the same disadvantages as do all fixed methods.

Consequently, adaptive methods emerged and they are still considered the most efficient approaches to XML-to-relational mapping. Their high performance features can be easily proven. An adaptive method starts with a fixed schema-driven mapping strategy which is modified so that the efficiency of query evaluation is enhanced. Hence, assuming that we start with the “best so far” fixed mapping strategy; the adaptive approach will provide at least an equally efficient mapping.
4.2. Overview of Adaptive Approaches

The existing representatives of adaptive methods can be divided into cost-driven and user-driven categories. Both approaches are based on the idea of exploiting additional user-provided information in order to appropriately adapt the target database schema. In the former case, it is extracted from a sample set of XML documents and/or XML queries which characterize the typical future usage. In the latter case, it is specified by user-provided annotations; i.e., the user directly specifies the required changes to a default mapping. Although there are many instantiations of the two approaches, there are still numerous weak points and open issues that should be improved and resolved.

Cost Driven Techniques

Cost-driven techniques can choose the best storage strategy for a particular application automatically, without any human intervention. Apart from various parameters of particular algorithms, the user can influence the mapping process through the provided XML schema, set of sample XML documents or data statistics, set of XML queries and eventually their weights. But after providing the input information, the algorithms cannot be influenced further by the user.

We can divide the existing cost-driven approaches into two types – single-candidate and multiple-candidate. Single-candidate approaches involve a straightforward mapping algorithm which provides a single output relational schema based on the input data. Multiple-candidate approaches also process the input data, but before providing the resulting relational schema, they evaluate multiple candidate solutions and choose the one which suits the considered application the most.

Single-Candidate Approaches

One of the first attempts at a cost-driven adaptive approach, also probably the only known representative of single-candidate approaches, was proposed in (Klettke et al., 2000). It is based on the observation that if XML documents are mostly semi-structured, a classical decomposition of unstructured or semi-structured XML parts into relations leads to inefficient query processing caused by a surfeit of join operations. Hence, the algorithm stores well-structured parts into relations using an analogy of the Hybrid algorithm and semi-structured parts using an XML data type, which supports path queries and XML-aware full-text operations. The main concern is to identify the structured and semi-structured parts of the input XML schema. The process consists of the following steps:

- A schema graph \( G \) (similar to usual DTD graph (Shanmugasundaram et al., 1999), where nodes represent elements, attributes and operators and edges represent relationships among them) is built for a given DTD.
- For each node \( v \in G \) a measure of significance \( \omega_v \) (see below) is determined.
- Each node \( v \in G \) which satisfies the following conditions is identified:
  - \( v \) is not a leaf node.
  - \( \omega_v < \omega_{LOD} \), where \( \omega_{LOD} \) is the required level of detail.
  - For each descendant \( d \) of \( v \) \( \omega_d < \omega_{LOD} \).
  - \( v \) does not have a parent node which would satisfy the conditions too.
Each subgraph of $G$ consisting of a previously identified node $v$ and its descendants is mapped to XML data type.

The remaining parts of the graph are mapped using a fixed schema-driven mapping strategy.

The measure of significance $\omega_v$ of a node $v$ is defined as:

$$\omega_v = \frac{1}{2} \omega_{S_v} + \frac{1}{4} \omega_{D_v} + \frac{1}{4} \omega_{Q_v}$$

where $D$ is a set of all given documents, $D_v \subseteq D$ is a set of documents containing $v$, $Q$ is a set of all given queries and $Q_v \subseteq Q$ is a set of queries containing $v$. $\omega_{S_v}$ is derived from the DTD structure as a combination of predefined weights expressing position of $v$ in the graph and complexity of its content model.

**Multiple-Candidate Approaches**

Each of the existing multiple-candidate techniques (Ramanath et al., 2003; Xiao-ling et al., 2003; Zheng et al., 2003) can be characterized by:

- an initial input XML schema $S_{init}$,
- a set of XML schema transformations $T = \{t_1, t_2, ..., t_n\}$, where $\forall i : t_i$ transforms a given schema $S$ into a schema $S'$,
- a fixed XML-to-relational mapping function $f_{map}$ which transforms a given XML schema $S$ into a relational schema $R$,
- a set of input sample data $D_{sample}$ consisting of a set of sample XML documents $D$ and XML queries $Q$ which characterize the future application and
- a cost function $f_{cost}$ which evaluates the efficiency of a given relational schema $R$ with regard to the set $D_{sample}$.

The required result is an optimal relational schema $R_{opt}$, i.e., a schema, where $f_{cost}(R_{opt}, D_{sample})$ is minimal. A naïve, but illustrative, cost-driven storage strategy that is based on the idea of “brute force” is depicted by Algorithm 1.

**Algorithm 1. A naïve search algorithm**

**Input:** $S_{init}, T, f_{map}, D_{sample}, f_{cost}$

**Output:** $R_{opt}$

1: $S \leftarrow f(S_{init})$
2: while $\exists t \in T \& s \in S$ s.t. $t(s) \notin S$ do
3:  $S \leftarrow S \cup \{t(s)\}$
4: end while
5: \( \text{cost}_{\text{opt}} \leftarrow \infty \)
6: \textbf{for all } \( s \in S \) \textbf{ do}
7: \( R_{\text{tmp}} \leftarrow f_{\text{map}}(s) \)
8: \( \text{cost}_{\text{tmp}} \leftarrow f_{\text{cost}}(R_{\text{tmp}}, \text{D}_{\text{sample}}) \)
9: \textbf{if } \text{cost}_{\text{tmp}} < \text{cost}_{\text{opt}} \textbf{ then}
10: \( R_{\text{opt}} \leftarrow R_{\text{tmp}} \)
11: \( \text{cost}_{\text{opt}} \leftarrow \text{cost}_{\text{tmp}} \)
12: \textbf{end if}
13: \textbf{end for}
14: \textbf{return } R_{\text{opt}}

The naïve algorithm first generates a set of possible XML schemas \( S \) using transformations from set \( T \) and starting from initial schema \( S_{\text{init}} \) (lines 1 – 4). Then it searches for schema \( s \in S \) with minimal cost \( f_{\text{cost}}(f_{\text{map}}(s), \text{D}_{\text{sample}}) \) (lines 5 – 13) and returns the corresponding optimal relational schema \( R_{\text{opt}} = f_{\text{map}}(s) \). It is obvious that the complexity of such an algorithm strongly depends on the set \( T \). It can be proven that even a simple set of transformations causes the problem of finding the optimal schema to be NP-hard (Xiao-ling et al., 2003). Thus, in practice, the techniques search for a suboptimal solution using various heuristics, greedy strategies, approximation algorithms, terminal conditions, etc. For instance, the FlexMap framework (Ramanath et al., 2003) as well as strategy proposed in (Zheng et al., 2003) optimize the naïve approach using a simple greedy strategy. On the other hand, the Adjustable and Adaptable Method (Xiao-ling et al., 2003) transforms the given problem to features of generic algorithms and it terminates either after a certain number of transformations or if a reasonably efficient schema is achieved.

The biggest set \( T \) of XML-to-XML transformations was proposed for the FlexMap framework and involves the following XSD modifications:

- **Inlining and outlining** – mutually inverse operations which enable storing columns of a subelement or attribute either in a parent table or in a separate table
- **Splitting and merging elements** – mutually inverse operations which enable storing columns of a shared element either in a common table or in separate tables, each for a particular sharer
- **Associativity and commutativity** – operations which enable grouping different elements into one table
- **Union distribution and factorization** – mutually inverse operations which enable separating out components of a union using equation \( (a, (b \mid c)) = ((a, b) \mid (a, c)) \)
- **Splitting and merging repetitions** – exploitation of equation \( (a^+) = (a, a^*) \)
- **Simplifying unions** – exploitation of equation \( (a \mid b) \subseteq (a?, b?) \)

Note that except for commutativity and simplifying unions, the transformations generate an equivalent schema, i.e., a schema with the same set of valid XML documents. Commutativity does not preserve the order of the schema, while simplifying unions generates a more general schema, i.e., a schema with a larger set of allowed document instances.

The process of evaluating \( f_{\text{cost}} \) is, in all the approaches, significantly optimized using cost estimation. A naïve approach would require construction of a particular relational schema, loading sample XML data into the relations and cost analysis of the resulting relational structures.
The *FlexMap* evaluation exploits an XML Schema-aware statistics framework *Statix* (Freire et al., 2002) which analyzes the structure of a given XSD and XML documents and computes their statistical summary. The XML statistics are then mapped to relational statistics regarding the fixed XML-to-relational mapping and, together with sample query workload, are used as an input for a classical relational optimizer for estimating the resulting cost. The other systems use a similar strategy and estimate the cost of the query using Markov tables or various statistical approaches.

**User Driven Techniques**

Undoubtedly, the most flexible adaptive approach to XML-to-relational mapping is “to leave the whole process in the hands of a user” who defines both the target database schema and the required mapping. We speak about so-called user-defined mapping techniques (Amer-Yahia, 2003). Probably due to simple implementation they are especially popular and supported in most commercial database systems. At first sight the idea is correct – users can decide what suits them most and are not restricted by features or disadvantages of a particular technique. The problem is that such an approach assumes that users are skilled in two complex technologies: relational databases and XML. Furthermore, for more complex applications, the design of an optimal relational schema is generally not an easy task.

On this account, a new type of adaptive mapping strategy, so-called user-driven methods, has appeared. The main difference is that the user can influence a default fixed mapping strategy using annotations which specify the required mapping for particular schema fragments. The set of allowed mappings is naturally limited, but still powerful enough to define various mapping strategies. In other words, the user helps the mapping process, rather than performing it directly. Each of the user-driven techniques is characterized by:

- an initial XML schema \( S_{init} \),
- a set of fixed XML-to-relational mappings \( f_{map}^1, f_{map}^2, ..., f_{map}^n \),
- a set of annotations \( A \), each of which is specified by name, target, allowed values and function and
- a default mapping strategy \( f_{def} \) for not annotated fragments.

The existing approaches can be also divided according to the strategy used to provide the resulting relational schema. We differentiate so-called *direct* and *indirect* mappings. Most of the existing approaches belong to the first group, where the user-specified annotations are just directly applied on the annotated schema fragments. The main idea of indirect approaches is to exploit the provided information as much as possible and to use it also for those parts of the schema that are not annotated.

**Direct Mapping**

The approaches exploiting direct mapping are based on a straightforward algorithm. Firstly, the initial schema \( S_{init} \) is annotated with user-required mapping strategies from \( A \). Secondly, the correctness of the annotations is checked. And finally, the XML schema is mapped. For annotated schema fragments, the specified mapping strategies \( f_{map}^1, f_{map}^2, ..., f_{map}^n \) are used; for schema fragments that are not annotated, \( f_{def} \) is used. The existing approaches differ from one another mainly in the supported mapping strategies.
The Mapping Definition Framework (Amer-Yahia et al., 2004) is probably the first representative of direct user-driven mapping strategies. It allows users to specify the required mapping and it is able to check correctness and completeness of such specifications and to complete possible incompleteness. The set of annotating attributes $A$ is listed in Table 1.

### Table 1. Annotation attributes for MDF

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Target</th>
<th>Value</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Outline</td>
<td>attribute or element</td>
<td>true, false</td>
<td>If the value is true, a separate table is created for the attribute/element. Otherwise it is inlined to parent table.</td>
</tr>
<tr>
<td>Tablename</td>
<td>attribute, element or group</td>
<td>string</td>
<td>The string is used as the table name.</td>
</tr>
<tr>
<td>Columnname</td>
<td>attribute, element or group</td>
<td>string</td>
<td>The string is used as the column name.</td>
</tr>
<tr>
<td>Sqltype</td>
<td>attribute, element or simple type</td>
<td>string</td>
<td>The string defines the SQL type of a column.</td>
</tr>
<tr>
<td>Structurescheme</td>
<td>root element</td>
<td>KFO, Interval, Dewey</td>
<td>Defines the way of capturing the structure of the whole schema.</td>
</tr>
<tr>
<td>Edgemapping</td>
<td>element</td>
<td>true, false</td>
<td>If the value is true, the element and all its subelements are mapped using Edge mapping (Florescu et al., 1999).</td>
</tr>
<tr>
<td>Maptoclob</td>
<td>attribute or element</td>
<td>true, false</td>
<td>If the value is true, the element/attribute is mapped to a CLOB column.</td>
</tr>
</tbody>
</table>

As we can see from the table, the set of allowed XML-to-relational mappings involves inlining and outlining of an element or an attribute, Edge mapping strategy and mapping an element or attribute to a CLOB column. Furthermore, it enables specifying the required capture of the structure of the whole schema using one of the following approaches:

- **Key, Foreign Key and Ordinal Strategy (KFO)** – each node is assigned a unique ID and a foreign key pointing to parent ID; the sibling order is captured using an ordinal value
- **Interval Encoding** – a unique $\{t_{\text{start}}, t_{\text{end}}\}$ interval is assigned to each node corresponding to preorder ($t_{\text{start}}$) and postorder ($t_{\text{end}}$) traversal entering time
- **Dewey Decimal Classification** – each node is assigned a path to the root node described using concatenation of node IDs along the path

Attributes can be considered as side effects for specifying names of tables or columns and data types of columns. Parts that are not annotated are stored using user-predefined rules, where such a mapping is always a fixed one.

The $XCacheDB$ system (Balmin et al., 2005) is the second existing representative of a direct user-driven strategy. The system also enables inlining and outlining of a node, storing a fragment into a BLOB column, specifying table names or column names and specifying column data types. The main difference is in the data redundancy allowed by attribute $\text{STORE\_BLOB}$ which enables to shred the data into table(s) and, at the same time, to store pre-parsed XML fragments into a BLOB column.
Indirect Mapping

Indirect mapping strategies try to exploit the user-provided information as much as possible. They are based on the idea that the annotations can be directly applied not only on particular schema fragments, but also for mapping the remaining schema fragments. The first and probably still the only representative of indirect user-driven strategies is system UserMap (Mlynkova, 2007). Its set of annotating attributes $A$ covers most of the features of the previous two systems. In addition, the UserMap authors specify three types of annotation intersections – overriding, redundant and influencing – which define the resulting mapping strategy in case multiple annotations are specified for a single schema fragment. In case of overriding annotations, only one of the respective mapping strategies is applied on the common schema fragment. In case of redundant annotations, both the mapping strategies are applied resulting in redundant storage. And finally, if the annotations are influential, they influence each other and the respective mapping strategies are combined, resulting in a single storage strategy.

The indirect mapping proposed in the system is based on a simple observation that the user-provided schema annotations can be viewed as hints about how to store particular schema fragments. And if we know the required mapping strategy for a particular schema fragment, we can assume that structurally (or semantically) similar schema fragments should be stored in a similar way. Hence, the system iteratively searches for similar schema fragments to find a more suitable mapping strategy.

4.3. Summary

To conclude this section, we provide an explanatory overview of the existing approaches to XML-to-relational mapping. Table 2 involves classification of the existing approaches, the key information exploited in the mapping process and respective sample representatives.

Table 2. Overview and summary of XML-to-relational mapping approaches

<table>
<thead>
<tr>
<th>Approach</th>
<th>Exploited information</th>
<th>Sample representatives</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fixed (schema-oblivious)</td>
<td>XML data model</td>
<td>Florescu et al., 1999</td>
</tr>
<tr>
<td>User-defined</td>
<td>Purely user-specified mapping</td>
<td>Amer-Yahia, 2003</td>
</tr>
<tr>
<td>Schema-driven</td>
<td>DTD-driven XML schema (DTD/XSD), respective constraints</td>
<td>Shanmugasundaram et al., 1999; Mlynkova et al., 2004</td>
</tr>
<tr>
<td></td>
<td>XSD-driven</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Constraints-preserving</td>
<td>Chen et al., 2003; Davidson et al., 2007</td>
</tr>
<tr>
<td></td>
<td>Multiple-candidate</td>
<td>Ramanath et al., 2003; Xiao-ling et al., 2003; Zheng et al., 2003</td>
</tr>
<tr>
<td></td>
<td>User-driven Direct XML schema, annotations</td>
<td>Amer-Yahia et al., 2004; Balmin et al., 2005</td>
</tr>
<tr>
<td></td>
<td>Indirect</td>
<td>Mlynkova, 2007</td>
</tr>
</tbody>
</table>

5. OPEN ISSUES AND POSSIBLE SOLUTIONS
As we can see from the previous overview, there exist plenty of approaches to XML processing in RDBMS. But, although each of the existing approaches contributes certain interesting ideas and optimizations, there is still room for possible future improvements to the adaptive methods. We describe and discuss them in this section and we also propose several solutions and possible enhancements.

5.1. State-of-the-Practice Challenges

A promise of XML is the smooth evolution of web document schemas and consequently their employment by applications that are aware of and leverage them for information manipulation and transformation. Considerable progress has been made in meeting the needs of data management practitioners in the past decade; but some challenges remain.

Better Decomposition

Methodologies are needed to address the need to evolve the storage and processing of XML data, particularly when it has been decomposed (i.e., “shredded”) into a relational database. Designers should note that supporting XML’s ordered data model is critical in many domains (e.g., content management) where order is meaningful and exploitable. They should confirm that the automation approaches they propose include built-in order encoding methods as part of the transformations that map XML documents into relational data models. This will ensure that sufficient information is captured to reconstruct ordered XML documents as needed, and to allow exploitation of ordering during query operations.

Developers also have recognized that shredding is not a one-time event. Logical and physical mappings can rarely remain static. Data evolve through time as information exchange requirements morph. The basis of XML is that transactions will contain both data and descriptions of what the data represents, thereby reducing the degree to which applications creating or receiving data need to be customized in order to interact with each other. But the mapping into relational tables can be “lossy” of this additional metadata.

Scalability

The logical mappings from an XML schema to a relational schema are also significant, especially when a paramount requirement is to ensure efficient query execution and meaningful, relevant results. Physical data breakdown and query performance as the database grows may even make or break the solution. Similarly, the combination of potential logical and physical design considerations – such as the selection of the physical structures of the database to which the XML is shredded, its consistency with the syntax and semantics of the original document, the degree to which the mapping is or is not “informed” by user input – produces an extremely large search space of possibilities. Choices within this space can have significant impacts on performance and efficiency and must be made with much care so that both anticipated – and reasonable though unanticipated – client needs can be met.

In addition, electronic commerce sites and content providers who rely on XML technologies increasingly need to deploy solutions where the data volume quickly grows to exceed “toy” size and whose content homogeneity cannot be guaranteed as stable, and which indeed changes
considerably, albeit not always consistently over time. Key considerations include strategies for avoiding reloading an entire database to store evolved data, and mitigating the need to change current processing software to handle the evolved data without compromising performance.

**Optimization**

Many of the optimization issues associated with SQL/XML are well-known, due to its basis in SQL which has literally decades of vendor effort behind it. Although the topic of optimizing XQuery queries is attracting much research attention, many problems are still unresolved. Because it is much younger than SQL, there is little experience as a foundational baseline. Some focus areas for optimization are element construction, user defined recursive functions, algebraic rewriting of XPath, and better cost models. A good summary of current research directions in XQuery optimization can be found in (Kader, 2007).

**Standardization**

Ultimately, a standards-based approach is required to ensure that solutions use XML and relational data consistently and in an interoperable manner. It should be noted that presently there is no horizontal (i.e., widely adopted) industry standard mapping of an XML document into a relational store; instead, vendors have been developing proprietary approaches to doing so, or handing mapping tools to users and leaving them to do the analysis. But pushing standardization into the access layer via XQuery and SQL/XML may obviate this concern. It would be interesting to convincingly substantiate this argument. The XQuery Update Facility (Robie, 2007) needs to come to fruition to provide applications with a flexible interface for sub-document changes.

**Other Future Directions**

As evidenced by the offerings of the major vendors, there are a variety of XML-enabled databases, XML query engines, XML services and native XML database that constitute a future vision, as XML documents continue to prevail as the fundamental unit of information exchange. Still there are significant, relevant and timely challenges and gaps with respect to XML and relational convergence. There are many fruitful avenues for further work, where XML and relational technologies must co-exist. These include the following:

- strategies for mitigating the “impedance mismatch” between orderless tables and inherently ordered and hierarchically structured XML content (Tatarinov, 2002);
- approaches for assessing the “sweet spot” where a collection of documents that are data-centric and highly amenable to relational mapping crosses over to become one that is document-centric (semi-structured or high natural language content) and therefore a less suitable mapping candidate; and what to do when content evolves to this point;
- better algorithms for automated but informed schema shredding, with specific suggestions for investigation directions that may yield improvements;
- “apples-to-apples” metrics for benchmarking performance among the various XML-enhanced relational databases and applications ala (Schmidt, 2002);
- investigating the extent to which pushing standardization into the access layer mitigates vendor interoperability mismatches;
content pattern and sequence recognition techniques, which have applications for biomedical and genetic algorithms.

Just as a relational database has a row in a table as its fundamental unit, XML-native databases (e.g., Tamino, XHive, Ipedo, NeoCore, Xyleme) treat an XML document as their fundamental unit. Interesting areas to explore here include:

- probably the most typical source of benchmarking XML data are repositories with fixed;
- state of XML-native database support across a mix of relational source data (e.g., migrating to the native store), and/or proposals for strategically accomplishing this;
- optimization issues/challenges in the emergent XQuery Update Facility; and
- storage/processing challenges/solutions for evolving XML Schemas and content.

5.2. State-of-the-Art Challenges

As in the realm of practice, so too in the research area are there promising and verified approaches, but we can still find various open issues and problems to be solved.

Problem of Missing Input Data

As we already know, the adaptive methods are based on the idea of exploiting various types of supplementary information. In particular, the set of input data usually consists of an XML schema $S_{init}$, a set of XML documents $\{d_1, d_2, \ldots, d_k\}$ valid against $S_{init}$ and a set of XML queries $\{q_1, q_2, \ldots, q_l\}$ over $S_{init}$. But, naturally, not always all the data are available.

The problem of missing input XML schema has already been outlined in the introduction in connection with the advantages and disadvantages of generic and schema-driven methods. Since we assume that adaptability is the ability to adapt to a given situation, an adaptive method which does not depend on the existence of an XML schema, but which can exploit the information if it is given, is probably a natural first type of improvement. A possible solution to the problem of missing $S_{init}$ is the exploitation of methods for automatic XML schema construction for the given set of XML documents (e.g., Moh et al., 2000; Vosta et al., 2008). These methods are able to infer corresponding content models from a given sample set of (similar) XML documents. In addition, since we can assume that XML documents are more precise sources of structural information, we can expect that a schema generated based on them will have good characteristics.

On the other hand, the problem of missing input XML documents can be solved at least partly using reasonable default settings based on general analysis of real XML data (e.g., Mignet et al., 2003; Mlynkova et al., 2006). Furthermore, the surveys show that real XML documents are surprisingly simple, so that the default mapping strategy does not have to be complex either. It should rather focus on efficient processing of frequently used XML patterns rather than all allowed constructs.

Finally, the presence of a sample query workload is crucial since there seem to be no analyses of real XML queries; i.e., there is no source of information for default settings as in the previous case. The reason is that the way to collect such real representatives is not as straightforward as in the case of XML documents, which easily can be trawled from the Internet. The best source of XML queries are currently XML benchmarking projects (e.g., Busse et al., 2003; Yao et al., 2003), but as the data and especially queries are supposed to be used for analyzing the
performance of a system in various situations, they cannot be considered as an example of a real workload.

Naturally, query statistics can be gathered by the system itself and the relational schema can be adapted continuously. But, this is the problem of dynamic adaptability which we will discuss later.

**Simplification of User Interaction**

Another related open problem arises from the fact that both the user-driven and cost-driven approaches require a relatively well-skilled user. In the former case, the user is expected to understand and specify the required mapping strategies; whereas in the latter case, the user must be able to specify exact XML queries. Hence, a natural question is whether we can simplify the process and make the system more user-friendly. We can assume that the user does not specify exact queries and mapping strategies, but rather typical operations with the data (together with their importance), such as exploiting/omitting data updates, a need for fast reconstruction of particular fragments, etc. The problem is how to state a reasonable set of such operations, i.e., a kind of language and to what extent they can reasonably influence the target mapping.

**More Efficient Search for Optional Mapping**

If we consider the way the existing cost-driven multiple-candidate approaches search for the optimal mapping strategy, we can identify a significant disadvantage. All the strategies use a kind of greedy search strategy which is based on the idea of improving the current solution until there exists a possible improving step. The problem is that such an approach can easily get stuck in a local optimum. In addition, if we consider the previously stated observation that the search algorithm should begin with a locally good schema, we can even increase the probability of this problem occurring.

The natural solution is to exploit an algorithm which adds a certain randomization to the search strategy. An example of a randomized search strategy can be a meta-heuristic called Ant Colony Optimization (ACO) (Dorigo et al., 2006) which enables searching also among less efficient solutions than the suboptimal one found so far.

**Combination of Cost-Driven and User-Driven Strategies**

Since both the cost-driven and user-driven approaches are able to store schema fragments in various ways, another natural improvement is their combination. Although this idea requires plenty of input information in the target application (i.e., schema annotations, sample queries and sample XML documents), the resulting mapping strategy can be expected to be much more suitable. In addition, we may again assume that the query statistics, as well as sample XML data, can be gathered and exploited by the system itself while the application is running. But, once again, this is the problem of dynamic adaptability.

**Theoretical Analysis of Related Problems**

As we can see from the overview of the existing methods, there are various types of XML-to-XML transformations, although the cited ones certainly do not cover the whole set of
possibilities. Unfortunately, there seems to be no theoretical study of these transformations, their key characteristics and possible classifications. Such a study could, among other things, focus on equivalent and generalizing transformations and as such serve as a good basis for the pattern matching strategy. Especially interesting would be the question of NP-hardness in connection with the set of allowed transformations and its complexity. Such a survey would provide useful information especially for optimizations of the search algorithm.

On the other hand, there already exist interesting theoretical studies focussing on different related problems. In particular, (Balmin et al., 2005) deal with the problem of data redundancy and related efficiency of query processing. They define and discuss four types of schema decompositions with regard to data redundancy and violations of normal forms. (Krishnamurthy et al., 2003) analyze theoretical complexity of several combinations of cost metrics and query translation algorithms. But, the analysis of the key aspect of adaptive strategies – the different storage strategies and schema transformations – is still missing.

Dynamic Adaptability

The last but not the least interesting open issue we will mention is naturally connected with the most obvious disadvantage of adaptive methods – the problem that possible changes to both XML queries and XML data can lead to an unacceptable level of degradation of efficiency. As already mentioned, related problems include missing input XML queries and ways of gathering them, as well as expecting the user to provide plenty of information on the future application a priori. Furthermore, the question of changes in the XML data itself opens another wide research area regarding updatability – a feature that is often omitted in current approaches although its importance is critical.

The need for a solution to these issues – i.e., a system that is able to adapt dynamically – is obvious and challenging, but it is not an easy task. The first related problem is how to find the new mapping strategy efficiently. Naturally, we could repeat the whole search strategy after a certain number of operations over the current relational schema has been performed, but this naïve strategy would be quite inefficient. A better solution seems to be the exploitation of a search strategy that does not have to be restarted when the related information changes; i.e., a strategy which can be applied on dynamic systems. An example of such strategy is the above-mentioned ACO meta-heuristic.

The dynamic system should also obviate the need for total reconstructions of the whole relational schema with the corresponding necessary reinsertion of all the stored data. Alternatively, such an operation should be done only in very special cases and certainly not often. On the other hand, this “brute-force” approach can serve as a good inspiration. It is possible to suppose that changes, especially in the case of XML queries, will not be radical but will progress gradually. Thus, changes in the relational schema will be mostly local and we can apply the expensive reconstruction just locally. However, we can again exploit the idea of pattern matching and try to find the XML pattern defined by the modified schema fragment in the rest of the schema.

Another question relates to how often the relational schema should be reconstructed. The logical answer is “not too often”, of course. But, on the other hand, research can be done on the idea of performing gradual minor changes. It is probable that such an approach will lead to less expensive (in terms of reconstruction) and, at the same time, more efficient (in terms of query
processing) systems. The former hypothesis should be verified; the latter one can be almost certainly expected. The key issue is how to find a reasonable compromise.

6 CONCLUSIONS

In this chapter we have described and discussed current state-of-the-practice offerings, state-of-the-art progress, and open issues related to XML data management in relational databases. We have viewed the problem from two perspectives – that of practice, and that of theory. Firstly, we provided an overview of existing approaches and methodologies and outlined their advantages and disadvantages. Then, we discussed many related open issues and, where possible, also their potential solutions.

Our aim was to show that, since the idea of processing XML data using relational databases is still a valid one, it can and should be further developed and refined. Although there already exist more efficient approaches – so-called native XML databases – the world of practice still prefers less efficient, but verified and reliable XML-enabled relational databases. In addition, with so much invested in relational tools, it is still more cost-effective to map XML content into them versus migrating legacy data into an XML-native database or simultaneously maintaining both relational and XML databases.

The short-term outlook is that the data accessed by legacy applications will increasingly come from XML sources, and that XML-aware applications will continue to increase in number and popularity; hence it will become more and more desirable to store XML natively and/or to map relational source data and queries into XML counterparts. In this chapter, we discussed the ways by which this is being done at present, as well as leading-edge research into how to improve current approaches. Observable trends suggest a transitional period where databases will contain both tables and shredded XML data, or tables and XML documents, with an increasing percentage of XML documents over time. During this transitional period, users will expect the choice of getting their results either as relational tuples or as XML, as required by consuming applications.

Web applications and XML documents will increasingly dominate the information exchange realm. Most web applications use XML to transfer data from the databases they access and vice versa. According to (Robie, 2003), XML applications that must use relational source data will choose from these approaches: apply SQL queries to the source, then transform the results to XML; use XML extensions provided by the vendor; use SQL/XML; use XQuery. Over the long haul, legacy relational stores will be sustained for archival purposes through their legacy SQL interfaces until this becomes impractical; at which time they will either be deprecated or refitted with XQuery-mediating middleware.

Finally, it is also important to mention that while all the open issues can be studied from various points of view, they are still closely interrelated and influence each other. Thus it is always important to consider each given problem globally and not overlook important side-effects and consequences. Similarly, although we have scrutinized the problem of XML data management in relational databases from two seemingly divergent perspectives, in fact both the practical and theoretical considerations converge to a common target – efficient and user-tailored processing of XML data.
ACKNOWLEDGMENT

This work was supported in part by Czech Science Foundation (GACR), grant number 201/06/0756.

REFERENCES


In ICDT’03: Proceedings of the 9th International Conference on Database Theory, pages 270 – 284, Siena, Italy. Springer-Verlag.


