Impedantix: An API for Native XML Data Stores

A. Bhm[1] M. Brantner[1*] S. Helmer[2] A. Hollmann[1] C-C. Kanne[1] N. May[1] G. Moerkotte[1]
[1]University of Mannheim [2]University of London
Mannheim, Germany London, United Kingdom
alex|msb|ah|cc|norman|moer@db.informatik.uni-mannheim.de sven@dcs.bbk.ac.uk

ABSTRACT

Database systems have to provide powerful Application Programming Interfaces (APIs) to facilitate the convenient development of data-intensive applications. While de-facto standards such as ODBC and JDBC have become widely accepted and adopted in the relational world, they provide only limited support for the specific requirements of XML data processing applications. To overcome these deficiencies, we (1) identify the specific requirements and (2) propose Impedantix, an API for interfacing with native XML data stores.

1. INTRODUCTION

XML has been adopted in a vast number of application areas, such as life sciences, geographic information systems, or distributed business processes. This success and the increasing volumes of XML data demand reliable and efficient ways to store and process them. As a result, native XML database systems (XDBMS) [1, 11, 20, 24] were developed to manage XML data. Moreover, the query languages XPath, XSLT, and XQuery, that allow to retrieve and transform XML data, were standardized.

Clearly, application programmers expect the application programming interfaces (API) for their favorite programming language both to incorporate powerful XML data management operations and to support convenient data retrieval using XML query languages.

For relational database systems, APIs such as ODBC [6] and JDBC [17] have evolved and have become de-facto standards for database interfaces. No such standards have converged for XDBMS, resulting in plenty of heterogeneous, vendor-specific programming interfaces [1, 7, 12, 20, 21, 25, 26]. The different approaches include extensions to relational interfaces [1, 20] as well as completely new API designs [26]. However, these existing solutions expose several deficiencies, as they include only rudimentary, text-based XML representations or lack essential features, such as physical storage management.

There is no single API design yet that fulfills the requirements for interacting with an XDBMS in an efficient and convenient manner.

*Supported by the Deutsche Forschungsgemeinschaft under grant MO 507/10-1.

Motivated by this observation, we summarize these requirements and propose *Impedantix*, a complete API design for XDBMS. We emphasize the need for an efficient and convenient application development by considering the peculiarities of XML processing, such as different data representations (e.g. textual, tree- or event-based).

Structure of this Paper. Before we elaborate on the functional requirements an XML API has to fulfill (see Sec. 3), we discuss the shortcomings of the existing approaches in the related work Sec. 2. After this, we describe the model of an XML data store which forms the basis of our API, Impedantix, in Sec. 4. Furthermore, we introduce and illustrate the concepts of the Impedantix API in Sec. 5. In Sec. 6, we present the custom view, a convenient approach for the efficient transformation of XML fragments into native programming language types. Finally, we conclude the paper in Sec. 7.

2. RELATED WORK

Our overview of related work starts with a review of several different concepts and APIs for accessing XML data in Section 2.1. Afterwards, we review both existing and proposed approaches for interfacing with database systems in Section 2.2. Section 2.3 outlines some of the various programming interfaces and APIs implemented in both commercial and open-source database servers and libraries.

2.1 Accessing XML

Most of today's programming languages do not support XML as a first class data type. However, many languages incorporate sophisticated string processing facilities. These facilities can be used to operate on the textual representation of an XML document. Unfortunately, this approach is not generally suitable for analyzing and modifying data based on the logical document structure. For example, even simple navigation operations on the logical document structure have to be represented by complex regular expressions. Thus, the document object model (DOM) [14] which provides programmers with a tree-based representation of an XML document and the event-based simple API for XML (SAX) [15] have become the most popular XML programming interfaces.

Apart from DOM and SAX, there are several other, language-specific APIs. Most of them provide tree-based document access and navigation such as JDOM [23] and dom4j [8].

Usually, modern programming languages incorporate some of the above APIs for operating on XML fragments. An example is the Java API for XML processing JAXP [27]. It provides DOM- and SAX-interfaces for accessing XML fragments and also allows using different parser implementations via a plugability layer. JAXP supports XPath query evaluation and XSLT transformations and

also includes the possibility to convert between the various supported XML representations (DOM, SAX, stream).

As an alternative to incorporating additional APIs, XML fragments can be transformed into the data types of the programming language e.g. Java objects (*marshalling*). The impedance mismatch between these different representations involves additional processing overhead and requires mapping definitions describing how XML fragments can be transcoded into native language constructs. However, this approach allows programmers to easily operate on XML data using native language primitives. The Java API for XML binding (JAXB) [18] and Castor XML [22] are examples of such databinding frameworks.

A third alternative to leverage programming languages for XML processing is the direct integration of native XML processing facilities. This can be done by incorporating a first-class XML data type and XML query languages such as XQuery [4] or XPath [2]. Microsoft's LINQ project [16] follows this approach. It incorporates XLinq, an API for native XML processing that is based on the data access features of Cω [3], into C# and Visual Basic. Similarly, the goal of XJ [13] is to integrate XML processing facilities and XML as a first-class type into Java.

Last but not least, we have to mention XQueryP [5]. XQueryP is an extension to XQuery and is designed to develop applications without relying on a host language. XQuery it extends XQuery allowing it to storing states in variables. Writing applications purely in XQueryP completely avoids the type system mismatch (of XML and the host language) and allows for global optimizations.

2.2 Database Interfaces

Within the last years, several interfaces for accessing relational databases from application programs have emerged. The Open DataBase Connectivity (ODBC) [6] and the Java Database Connectivity (JDBC) [17] are prominent examples. As these interfaces are designed to provide SQL-based database access, they do neither support XML query languages nor provide a native XML data type. To overcome these deficiencies, SQL/XML has been proposed [9], providing additional functions for handling XML data and supporting XML queries within SQL statements. The following example shows how an XQuery statement can be executed in SQL/XML.

```
PreparedStatement stmt=connection.prepareStatement(
  "select XMLQUERY('for $b in $booklist/book return
                    $b/title' PASSING ? AS \"booklist\"
                    RETURNING CONTENT NULL ON EMPTY)
                    AS result from BOOKS_XML")

ResultSet rs=stmt.executeQuery();
```

This approach of wrapping XML query functionality in SQL allows existing interfaces to be leveraged. However, as in the example above, it often requires cumbersome parameter passing between SQL and XML query statements (the auxiliary "booklist" variable is required as the "?" placeholder must not be contained in the XQuery statement). Additionally, special characters have to be escaped in nested queries and NULL values have to be handled carefully. To overcome these drawbacks, new interfaces and APIs have been proposed to interact with native XML databases.

The XQuery API for Java (XQJ) [10] is a joint effort of IBM and Oracle to provide a standardized interface to access XML database systems from Java applications. Being closely related to the SQL-specific JDBC, its goal is to provide similar functionality and interfaces for executing XQuery statements. As in JDBC, query results are represented as sequences. XQJ includes the possibility to access elements using (among others) DOM, SAX and text-based interfaces. Additionally, it supports user-defined transformation handlers, that can e.g. be used to convert XML data to Java objects.

The goal of the XML:DB API [26] project is to provide "a common access mechanism to XML databases", offering functionality equivalent to JDBC or ODBC for XML data stores. XML:DB proposes a hierarchy of arbitrary nested collection objects to allow the logical grouping of documents. Similar to the fragment/view concept in Impedantix, it uses a uniform representation for XML fragments called XMLResources. They allow accessing the represented data using DOM, SAX and textual interfaces. As Impedantix, the XML:DB API also includes a transaction service specification. However, it lacks several essential features such as physical storage management and XQuery support.

2.3 Implementations

Today's native XML database systems support the XML processing APIs and database interfaces described above to various extends. A common approach is to extend existing, relational database interfaces such as JDBC by introducing new (proprietary) XML types (e.g. [1, 20]). These XML types either represent XML documents in textual format [20] or allow accessing them with XML-specific interfaces such as DOM and applying XSLT transformations [1].

Several implementations including Apache Xindice [12] and Tamino [25] adopt the XML:DB API specification. Others choose to implement proprietary user interfaces such as Berkeley DB XML [21]. It provides a combined physical and logical repository view, where all document containers are physical files, thus disallowing nesting of containers. BerkeleyDB XML provides DOM-based document access and also includes a proprietary, tree-based interface.

3. FUNCTIONAL REQUIREMENTS

Having discussed the features and shortcomings of the existing approaches, we now summarize the key requirements an industrial-strength solution has to fulfill.

Database Management. XML database management systems provide facilities to organize and maintain persistent XML data collections. Conceptually, data management can be divided into two components.

Logical data management is used to associate XML documents with logical organizational units, e.g. by inserting related documents into a conjoint collection. By aggregating document collections in turn, logical hierarchies can be created within the data store. To provide these features, an API needs to support importing, deleting, and managing documents as well as logical collections.

Physical management is used to assign *physical storage capacity* (e.g. provided by storage media such as hard discs) to the logical components. While the DBA is usually responsible for physical data management, a seamless integration of them is desirable. By finding a common abstraction and interface for logical and physical data management, application code becomes more reusable and portable.

Data Access. There are several different interfaces for accessing XML data, including textual, event- [15] and tree-based representations [8, 14, 23]. The right interface to choose depends on the individual requirements of an XML application. Application programmers should have full freedom to choose their most appropriate interface for accessing XML data. As a result, seamless and consistent support for *all* of these representations is an important requirement to reduce the effort for moving to a more appropriate representation.

Queries. Querying data is one of the most important features of a database system. The World Wide Web Consortium proposes a processing model for querying XML (see [2, 4]). This processing model also applies to queries embedded into a host programming language. It is reasonable to adhere to this processing model in an API, as it considers other XML-specific standards, such as data model generation, schema import processing, and serialization. Since this processing model is shared by several XML query and transformation languages, programmers should be able to easily switch between them, e.g. from XPath or XSLT to XQuery.

Transactions. Advanced applications allow multiple users to access XML data concurrently and modify the content of the XML data store, e.g. they bulkload new documents or execute an XQuery Update statement. Hence, it is natural to expect that the XDBMS guarantees ACID properties for operations. To provide these properties almost all database systems rely on the concept of transactions. Thus, an API has to incorporate transactional processing facilities, allowing developers to perform essential operations such as starting, committing, and aborting transactions. Additionally, advanced features that might be provided by the underlying XML dat a store such as different (transaction-specific) isolation levels, checkpointing, savepoints and distributed commit protocols [19] have to be supported by the API.

Native Language Types. Interactions between database systems and application programs involve additional complexity if there is a mismatch between the type system of the database system and the programming language. For example, this *impedance mismatch* is encountered when trying to serialize objects from object-oriented languages to relational database systems. Application developers face this problem when trying to handle XML data stored in a database system, as — despite recent efforts [13, 16] — many popular and wide-spread programming languages (including Java, C, C++ and C#) do not support XML as a first-class data type.

Hence, a convenient and robust transformation of XML data into first-class data types of the host language is another key requirement for an API. Most of the existing transformation solutions (such as [18, 22]) rely on auxiliary application code and mapping definitions. To avoid this additional development overhead, the API should incorporate a simple-to-use and declarative mechanism to specify such a transformation, thus allowing developers to convert XML documents to e.g. Java or C++ objects. In particular, it should be possible to transform single literals into objects or multiple inputs nodes into sets of objects, respectively.

4. MODEL OF AN XML DATA STORE

In the previous section, we identified two aspects of data management. Logical data management allows for the hierarchical maintenance of documents based on logical coherence in application programs. It is complemented by the physical data management component, that allows to control physical parameters such as assigned disk capacity or data placement on storage media.

By separating physical data management aspects from logical document access, application programmers can abstract from the physical placement of data. This allows them to solely operate on the logical data hierarchy once the physical infrastructure has been defined.

4.1 Logical Data Management

The logical data management of Impedantix uses a hierarchical, tree-based model to manage and group XML fragments (see Fig. 1). Its topmost organizational unit is the *Instance*, representing an entire database system. An instance consists of a set of *Repositories* that are used to manage documents belonging to a particular application domain. As an example, a bookstore could use distinct repositories for its book inventory, the customer database and accounting information, respectively. Each repository contains a set of *Collections*. A collection represents a group of logically related documents, typically sharing a conjoint schema. For example, the various genres of a book inventory repository can be represented by corresponding collections. Each individual XML document stored in the database is represented by a *Document*.

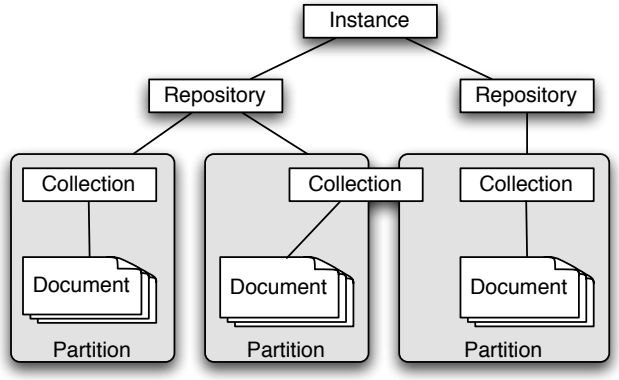

Figure 1: Impedantix model of an XML data store

4.2 Physical Data Management

Physical data management allows to control physical database characteristics, such as the placement of data on local or remote file systems. Impedantix applies the concept of storage containers (called *Partitions*) that can be assigned to logical entities, such as a collection or a repository. Partitions can be shared among different collections, and several partitions can be associated to a particular collection at the same time (see Fig. 1). Apart from handling partitions, physical data management is also used for defining storage capacity and physical locations of system facilities such as the recovery log.

Notice that physical data management focuses on the database server and, thus, is usually not available in the API. However, in an embedded XML application, as it is targeted by Impedantix, the XDBMS runs in the address space of the application program. Thus, it is natural to include the physical data management into the API. Furthermore, we argue in this paper that a seamless integration of the data management is easily possible.

5. THE IMPEDANTIX API

After defining the requirements and introducing the logical model of an XML Data Store, we now present the Impedantix API.

We demonstrate the API concepts fulfilling the requirements from Sec. 1 by means of code snippets. A ready-to-compile example containing these snippets can be found on the Natix website [1].

All concepts of the Impedantix API can be implemented by various object-oriented programming languages. In this paper, we stick to C++ for explaining the API design.

Impedantix consists of three major concepts: (1) *Requests* are used to process operations, (2) *Fragments* are used as an opaque

[1] http://db.informatik.uni-mannheim.de/natix.html

abstraction for any data, and (3) *Views* provide a flexible way to access the data represented by fragments. We elaborate on each of these concepts in one of the following sections, respectively.

5.1 Requests

The Impedantix API uses *Requests* to process a specific action. All actions are performed by initializing request objects and forwarding them to the database system. The system accepts these objects and executes the corresponding operations. Initialization is accomplished by passing parameters to the requests' constructors. Requests can either operate on the instance level (e.g. for creating a new database instance) or on the level of a single transaction (e.g. document import). Hence, we distinguish two types of requests: (1) instance-level requests and (2) transaction-level requests.

Instance-Level Requests. An Instance represents the entire database system, i.e. it is a set of partitions and configuration parameters. Application programs address a physical (on disk) instance using a main-memory instance object. For performing an instance-level operation, a request object can be sent directly to such an instance object. This is done by calling the process method of the instance class with the request object as a parameter. For example, in the following first code snippet, a request object for creating a (new) physical instance is initialized and then forwarded to an instance object named `inst`.

```
Instance inst;
CreateRecInstance createinst("bookstore", 8192,
                             20000, false);
inst.process(createinst);
```

The system accepts this request with its parameters and executes it. In this particular case, the system creates a new recoverable instance named bookstore on disk. It has a page size of 8KB and is associated with a recovery partition with a capacity of 20,000 pages. If a homonymous instance already exists, the boolean parameter `false` indicates that the existing instance should be overwritten.

It is important to note that the above call only creates a new instance. Before an application is actually able to operate on the instance, a startup request must be issued to connect the main-memory instance object to the physical instance, i.e. initialize main-memory data structures and buffers.

```
Startup startup(inst, "bookstore");
```

Additionally, the snippet shows that by passing the instance object as first parameter to a request, the process method is called implicitly, thus saving to write a line of code.

Many other operations can be performed on the instance level. For example, there exist requests to backup instances or manage access control for particular operations or instances.

Transaction-Level Requests. Impedantix allows to atomically perform a series of logically related operations - such as importing and querying a document - by executing them in the context of the same transaction.

Transactions are started by instantiating the Transaction class, using the Instance as a parameter. A transaction is active as long as the transaction object exists and commit or abort have not been called. By default, transactions are considered to be update transactions.

In the following example, we initiate a new transaction and create a partition named `Customer Data` in the file `Customer Data.part` with a size of 10,000 pages.

```
Transaction trans(inst);

CreatePartition part("Customer Data", "file",
                     "Customer Data.part", 10000);
trans.process(part);
trans.commit();
```

Processing requests in transactions is handled similarly to processing instance-level requests. Specifically, a transaction-level request is executed by calling the `process` method of a transaction object passing the corresponding request as parameter. Finally, by calling commit, the transaction ensures that changes safely reach stable storage.

When creating a new transaction, a number of optional parameters can be specified, e.g. for indicating that a transaction only requires read access, thus potentially achieving better concurrency. Additionally, the isolation level required by the particular transaction can be specified. In the following example, we create a read-only transaction requiring isolation level three and immediately abort the corresponding transaction.

```
Transaction readTrans(inst, READONLY, LEVEL3);
//perform some read operations
readTrans.abort();
```

The underlying data store has to make sure that the requested isolation levels can be provided, for example by reflecting Impedantix requests by corresponding locks. Depending on the requested isolation level, this can be implemented by acquiring short-duration locks for single database requests (e.g. read-operations). Other operations, such as document updates, might require long-duration locks held until the end of the transaction or medium duration locks spanning multiple requests (e.g. for providing cursor stability when accessing a document collection).

Exception Handling. Reacting to run-time errors that occur during the execution of an action is, of course, another important feature an API has to incorporate. For example, a parse error during document import must be caught and be treated adequately. In any case, it is important to handle and appropriately react to errors. For this reason, most object-oriented languages provide the concept of exceptions. In the Impedantix API, exceptions can be thrown when processing actions, i.e. requests. For instance, when trying to create a partition on a physical medium that has not enough space available, an `InsufficientSpace` exception is thrown.

5.2 Fragments

Similar to the concept of a file in many operating systems (e.g. Unix), Impedantix provides an abstraction for data. In Unix, applications can operate on files without being aware of whether the file is a device, a pipe, a local file on the hard disk, or some remote file on a server. We call an abstraction of data a *Fragment*. Fragments may represent documents in the file system, documents stored in the data store, repositories, document collections, or query results. In the Impedantix API, fragments are referred to by instances of the class `FragmentDescriptor`.

In the following, we elaborate on the usage of fragments and their descriptors. We do this along with examples that show the management of repositories, collections, and documents.

Managing Repositories and Collections. Repositories are used to manage documents which belong to a particular application domain. In the following, we show how to create a new repository for storing our customer data.

```
CreateRepository createRep(trans, "Customer Data", part);

FragmentDescriptor cRep = createRep;
// use fragment descriptor

CloseFragment closeRep(trans, cRep);
```

The `CreateRepository` request takes the name of the repository and a partition in form of the `CreatePartition` request from one of the last snippets as parameters. The new request object is executed by passing it as a parameter to the transaction object's process method. Note that in the snippet above the transaction object is passed as first parameter to the request object. This invokes the process method automatically and is similar to executing instance-level requests.

If a request has a result, e.g. the opened repository referred to by a fragment descriptor, this result is stored in the request object's members. In the special case that a request has only one return value, a conversion operator is provided which allows to use the request object directly in place of the return value. For example, in the above snippet, it is used for assigning the newly created repository to the fragment descriptor `cRep`.

After the repository is no longer used, its fragment descriptor must be closed to free all required resources and release acquired transaction locks. This is done by passing the `CloseFragment` request to the transaction that was responsible for opening the fragment descriptor.

Similarly, we can create a new collection "Play" within the already existent product repository (`pRep`). Again, this is done by passing it a fragment descriptor of the opened repository as parameter.

```
CreateCollection createColPlays(trans, pRep, "Play");
```

Managing Documents. Fragments are not only an abstraction for partitions, repositories, or collections, but also for XML documents. Fragments are an abstraction for any representation or storage location. Hence, for example, a document can be referred to by a fragment descriptor, no matter whether it is represented as a main memory DOM tree, it exists in the database, or as a file in the file system.

The following example shows how to open a document that is stored in a file and, subsequently, store it in the database.

```
FragmentDescriptor in =
                OpenFileDocument(trans, "Hamlet.xml");

StoreDocument sDoc(trans, "Product Catalog", "Play", in);

CloseFragment closeInput(trans, in);
CloseFragment closeSDoc(trans, sDoc);
```

Depending on the specific source represented by the input fragment descriptor (i.e. a document in a file system in the above example), the underlying data store is free to choose the appropriate import method. For example, this could be a stream-based parser for text-based documents, a handler traversing a main-memory DOM source, or even a direct copy if the input source is already in the XML data store, thus avoiding additional parsing overhead.

5.3 Views

The right programming interface to choose for accessing XML data depends on the individual requirements of each application. It might be beneficial to access an XML document using a DOM interface in one application scenario (e.g. when updating individual nodes in a document), while an event- or text-based representation might be more suitable in other cases (e.g. when sending an XML document over a network).

To support the ability of choosing any kind of access method for any kind of data, Impedantix uses the concept of *Views*. Specifically, each view type represents a method for accessing the data represented by fragments.

5.3.1 Accessing Documents

The `OpenView` request is used to open a specific view class (e.g. `DOMView`, `SAXView`, `StreamView`, `SequenceView`, or `MetadataView`) on an arbitrary fragment. It takes the view class as template parameter and the fragment to access as parameter to the constructor.

In the following example, the `DOMView` is used for accessing a single document as a DOM tree.

```
OpenRepositoryDocument hamlet(trans, "Product Catalog",
                              "Play", "Hamlet.xml");

OpenView<DOMView> domView(trans, hamlet);
DOMNode *node = domView.getView()->getNode();
// ...
// Access the node using Xercesc 2.7's DOM interface
// ...
CloseView closeView(trans, domView);
CloseFragment closeDoc(trans, hamlet);
```

The `OpenRepositoryDocument` request provides a fragment descriptor for a document stored in the XML data store. The `OpenView` request then provides access to the resulting fragment in form of the `DOMView` class. This class, in turn, has a method for accessing the root node of the document in form of a `DOMNode`. At the end, the View and the opened document must be closed using the requests `CloseView` and `CloseFragment`, respectively.

5.3.2 Accessing Sequences

In addition to single documents, applications may also want to access sequence-valued data such as all collections in a repository, or all documents in a collection. Therefore, the Impedantix API incorporates sequence views such as the `DocumentSequenceView`, that provide iterator-based interfaces. For example, printing the names of all documents within a specific collection can be done as illustrated in the following snippet.

```
class Person
{
  Person(std::string name, std::string address);
  std::string theName, theAddress;
};

...

  DOMNode* addressbook = document->getDocumentElement();
  std::list<Person> adrs;
  for(DOMNode* person = addressbook->getFirstChild();
      person; person = person->getNextSibling())
  {
    if(person->getName()=="contact")
    {
      DOMNode *name;
      for(name=person->getFirstChild(); name;
          name = name->getNextSibling())
        if(name->getName()=="name")
          break;
      for(address=person->getFirstChild(); address;
          address = address->getNextSibling())
        if(name->getName()=="address")
          break;
      Person p(name->getValue(), address->getValue());
      adrs.push_back(p);
    }
  }
```

5.4 Automatically Releasing Resources

Every resource must be closed when it is not needed any more. Manually keeping track of opened fragments and views and closing them is a complex and error-prone task. To make applications more succinct, less error-prone, and easier to write, the Impedantix API provides the concept of *tidy* requests for automatically closing fragment descriptors. For every Request and `OpenView` class, there exists a variant with the suffix "Tidy" appended to the class name. The tidy variant guarantees that all occupied resources are released once the control flow reaches the end of the request's or view's scope.

For example, the snippet we presented to create a repository could be simplified, yielding the following

```
{
  CreateRepositoryTidy
        createCustomerRep(trans, "Customer Data", part);

  FragmentDescriptor customerRep = createCustomerRep;
  // use fragment descriptor
} // the FragmentDescriptor is closed automatically
```

In this case, the fragment descriptor is closed automatically when the control flow reaches the end of the scope, i.e. the closing right parenthesis.

This technique is particularly suited to C++ exception handling. Suppose an exception is thrown while importing a document, and it is not handled by a catch block on this level, but by a catch block in some caller, as shown in the following:

```
void importFunction() {
  OpenFileDocumentTidy oFile(trans,
                             "Hamlet.xml");

  // exception must be catched by caller
  StoreDocumentTidy sDoc(trans, "Product Catalog",
                         "Play", oFile);
}
```

In this case, the automatic destruction of two request objects shown above will prevent resource leaks without requiring the developer to write a local catch block.

5.5 Queries

Besides managing the physical and logical schema, it is an important feature to execute queries on arbitrary XML data stored in a database. Therefore, our API implements the processing model that is defined in the XQuery 1.0 [4] recommendation. This processing model is divided into two phases: the static analysis phase and the dynamic analysis phase. The realization of each of this phases is discussed in one of the following paragraphs.

The API concepts of Impedantix are used for query processing. Specifically, we use requests for processing actions (e.g. preparing or executing the query), fragments for representing query results, and views to access these results. To provide the opportunity of executing different query languages (e.g. XQuery, XQuery Update, or XPath), every request is passed a template parameter deciding on the particular query language. For example, in a first step, we create an in-memory object referring to an XQuery query as follows:

```
QueryHandle<XQuery> query =
  CreateQueryTidy<XQuery>(trans,
    "for $i in //PERSONA/name return <pers>{$i}</pers>");
```

Note that all resources that are allocated by the request are automatically released by using a tidy request. To make the code snippets easier to read, we will only use tidy requests in the following.

Static Analysis Phase. During the first phase of the processing model, the query is parsed, normalized, and a static type is assigned to each expression. Our API provides a *Static Context* that may contain information that is required during the static evaluation phase. Such a context is created using the `CreateStaticQueryContext` request. This request takes the query handle as parameter and automatically assigns the newly created static context to the query.

```
CreateStaticQueryContextTidy<XQuery> sCtx(trans, query);

PrepareQueryTidy<XQuery>(trans, query);
```

Based on the static context, the `PrepareQuery` request completes the static analysis phase and makes the query ready to execute.

Dynamic Analysis Phase. During the dynamic analysis phase, the query is evaluated. Analogously to the static analysis phase, a context (called *Dynamic Context*) is provided to the dynamic phase. The dynamic context contains the information that is needed during evaluation, e.g. the document on which the query should be evaluated. In our API this can either be a document or a collection. Hence, there exist constructors for creating a dynamic context taking either of them as parameters. Moreover, variables used in the query can be set in the context object. In the following snippet, for example, the query is executed on a document `Hamlet.xml`, which we have imported previously.

```
CreateDynamicQueryContextTidy<XQuery> dCtx(trans,  query,
                                "Product Catalog", "Play",
                                "Hamlet.xml");

ExecuteQueryTidy<XQuery> qResult(trans, query, dCtx);
```

The most suitable approach for query evaluation depends on the type of the data source represented by the fragment descriptor. For example, the underlying data store might choose a stream-based query evaluation if the input document is stored as a text file. Alternatively, it might pick a query execution engine performing navigational access if the data is stored in the database system. To enforce the use of a particular query execution engine, application developers may refine the corresponding template parameter (e.g. by choosing "XQueryStreaming" instead of the more general "XQuery").

Executing queries is a frequent operation when working with data that is stored in a database. To abbreviate the query execution process, there are a number of overloaded constructors for every input parameter combination, offering the possibility to skip some of the above steps. As in the following example, we pass the query as well as the collection on which it should be executed to the `ExecuteQuery` request. This immediately causes the query to be prepared and executed on the specified collection.

```
ExecuteQueryTidy<XQuery> qResult(trans,
                "/PLAY/TITLE[contains(., 'Hamlet')]",
                "Product Catalog", "Play");

OpenViewTidy<DocumentSequenceView> sView(trans, qResult);
```

The iterator of the `DocumentSequenceView` provides access to all documents for which the result of the query is not empty.

6. CUSTOM VIEW

The view concept of Impedantix allows application developers to choose from several XML processing APIs (e.g. DOM and SAX), picking those most suitable for their particular tasks. However,

there usually remains an impedance mismatch between the type systems of the programming language and the XML format. As a consequence, a number of transformation steps are required to convert between the language type system and the XML data representation before being able to process XML data using native language constructs (e.g. Java/C++ objects).

One possibility for performing such a transformation is to traverse the DOM representation of the document and create corresponding objects. The following code sample illustrates this process by extracting all person elements from an XML document and storing them as a list of C++ objects.

```
class Person
{
  Person(std::string name, std::string address);
  std::string theName, theAddress;
};

...

  DOMNode* addressbook = document->getDocumentElement();
  std::list<Person> adrs;
  for(DOMNode* person = addressbook->getFirstChild();
      person; person = person->getNextSibling())
  {
    if(person->getName()=="contact")
    {
      DOMNode *name;
      for(name=person->getFirstChild(); name;
          name = name->getNextSibling())
        if(name->getName()=="name")
          break;
      for(address=person->getFirstChild(); address;
          address = address->getNextSibling())
        if(name->getName()=="address")
          break;
      Person p(name->getValue(), address->getValue());
      adrs.push_back(p);
    }
  }
```

Obviously, writing this transformation code is a tedious process. To simplify application development, the custom view of Impedantix incorporates a more comfortable mechanism to transform XML fragments into first-class data types of the programming language. For example, the above transformation code can be substituted with the following, much shorter list of statements.

```
Matcher<std::list<Person>> matcher(
  collect("//contact",
    construct<Person>(extract<std::string>("name"),
                      extract<std::string>("address")),
    &std::list<Person>::push_back));

FragmentDescriptor addressbook = document;
std::list<Person> adrs=matcher.eval(addressbook);
```

The custom view uses native language constructs and a fixed set of predefined operators that can be arbitrarily combined for data extraction. As a result, no additional mapping definitions, preprocessing steps or auxiliary coding that are usually required by marshaling solutions (e.g. [18, 22]) become necessary.

In this section, we describe the concepts of the custom view, using its C++ binding as an example. Afterwards, we introduce the key operators provided and show how they can be used for mapping XML data to C++ objects.

6.1 Concepts

The custom view uses a *mapping function* for transforming XML elements into native language data types. The application developer defines this mapping function by combining a set of *operators*. Each operator represents a function that transforms a set of input XML nodes to a set of objects.

To allow the operators to perform the transformation functionality, each operator includes an `eval` function. Thus, the transformation defined by the top-level mapping function can be initiated by calling its `eval` function. The transformation function takes an input element represented by a fragment descriptor as parameter. To remain type safe, the operators provide template parameters that allow the specification of their return type.

The operators have a tree-based view of the XML document, allowing to express tree patterns using XPath expressions. This declarative way of extracting information from the XML document makes the approach of the custom view robust against schema changes. If the schema of a processed document changes, it is sufficient to adapt the corresponding XPath expression. Depending on the structural modifications performed by the schema update, there might even be no changes required at all (e.g. when some new elements get inserted).

By using a declarative, operator-based specification, the underlying data store is free to choose the optimal strategy for acquiring those parts of a document that are required by the custom view. For example, data could be retrieved by using a stream-based parser on a text file, by directly navigating a DOM tree, or by using the query execution engine for XML fragments stored in the database system (as described in Section 5.5).

6.2 Operators

The transformation functionality of the custom view is provided by a set of operators that can be combined to instantiate any (user-defined) C++ class, including container classes such as the list of the standard template library. The following subsections give a brief overview and some examples of the most important operators.

6.2.1 Extract

A fundamental operation is to extract single literals (e.g. strings) from a document. This functionality is provided by the *extract* operator. In the following example, it is applied to retrieve the content of the author element from an XML document.

```
FragmentDescriptor book = document;
Matcher<std::string>
    matcher(extract<std::string>("/book/author"));

std::string author = matcher.eval(book);
```

First, a *matcher* is specified. It represents a sequence of operators that should be evaluated in the context of an XML fragment. In the above example, this sequence consists only of a single extract operator. Its parameter is an XPath expression ("/book/author"), identifying the target node that should be extracted. A template parameter describes the type of the return value (std::string) that is used for both the matcher and its member operator. To actually retrieve a node from a given XML fragment, the matcher is evaluated on the input document by calling its `eval` operation with an input fragment as parameter. For programming convenience, the explicit definition of a matcher can be omitted by directly invoking the `eval` function of the (top-level) operator as in the following examples.

6.2.2 Collect

While the extract operator can be used to retrieve a single element from an XML document, the *collect* operator allows to retrieve multiple nodes. Its purpose is to perform set-based operations such as creating a list of objects from a sequence of XML elements.

```
list<std::string> titles = collect("//reference",
    extract<std::string>("title"),
    &std::list<std::string>::push_back).eval(book);
```

In the example above, the collect operator is used to retrieve a list of strings containing the titles of all references in a book. It obtains a sequence of all "reference" nodes using an XPath expression and invokes the extract operator for each candidate node in the sequence to extract its "title" child element. The collect operator invokes a user-defined operator for each encountered node. In our example, this operator is a function parameter (`push_back`) appending each title which was previously extracted to the list of results.

6.2.3 Construct

The *construct* operator can be used to instantiate a new object. Basically, construct invokes the constructor of a C++ class, passing parameters obtained from an XML document to it. In the example below, the extract operator is used twice to retrieve the string value of XML nodes, passing them to the construct operator to create corresponding C++ objects. Again, the collect operator is used to create a list of results.

```
// Author(std::string fName, std::string lName);
list<Author> citedAuthors = collect("//reference",
  construct<Author>(extract<std::string>("firstName"),
                    extract<std::string>("lastName")),
  &std::list<Author>::push_back).eval(book);
```

6.2.4 Operator Lifting

To increase the expressive power of the custom view, native language operators such as "+" or "*" can be "lifted" by defining them for the custom view constructs. As all custom view operators have a fixed type defined by a template parameter, language types and custom view operators can be used as operands for these lifted operators. Lifted operators may perform different operations, depending on the type of the operands. Thus, a "+" operator may perform string concatenation or add two numbers, e.g. `"SALE: " +extract<std::string> ("title") + "!"` or `extract<int> ("age") + 1`.

The following snippet shows the application of a lifted "*" operator. It is used to add the VAT to the price previously extracted from an XML document before passing it to a construct operator for object instantiation.

```
// Book(std::string bTitle, float bPrice);
FragmentDescriptor books = document;
list<Book> booklist = collect("//book",
    construct<Book>(extract<std::string>("title"),
                    extract<float>("price") * 1.07),
    &std::list<Book>::push_back).eval(books);
```

Lifted language operators usually allow for both more convenient application development and faster application execution. In the above example, the "*" operator saves application developers from iterating over the collected booklist for adapting prices.

6.2.5 Cond

The *cond* operator chooses from one of two different operators based on the return value of an XPath statement. In the example below, it is used to determine whether a reference element in an XML fragment has both a "month" and "year" child. If both nodes exist, the extract operator is used to retrieve their string values before the results are concatenated using lifted "+" operators. If there are no such child elements, the string value of the "date" element is extracted. Again, results are accumulated using the collect operator and a C++ list.

```
std::list<std::string> allDates = collect("//reference",
    cond("date[month and year]",
    extract<std::string>("date/month") +" "
        +extract<std::string>("date/year"),
    extract<std::string>("date"))
  &std::list<std::string>::push_back).eval(addressbook);
```

7. CONCLUSION

We presented Impedantix, a full-featured API for the efficient and convenient development of XML processing applications interacting with an XML Data Store.

We derived the requirements the Impedantix API has to meet. To this end, we introduced — based on a model for logical and physical data management — the concepts of (1) Requests for executing actions, (2) Fragments as an abstraction of data, and (3) Views for accessing the data represented by Fragments.

Furthermore, we introduced a special view, called Custom View. It can be used to compensate the impedance mismatch between XML data fragments and the type systems of programming languages which do not provide an XML data type. In contrast to most other marshalling solutions, the Custom View does not require auxiliary mapping definitions or preprocessing steps and also allows developers to easily deal with schema changes.

In order to prove its practicability, the Impedantix API is implemented in the Natix XML Data Store C++ library. It is available on the Natix website (http://db.informatik.uni-mannheim.de/natix.html), which additionally contains a link to a detailed manual and class documentation.

Acknowledgments. We would like to thank Simone Seeger for her comments on the manuscript.

8. REFERENCES

[1] D. Adams. Oracle XML DB developer's guide, 10g release 2 (10.2). Technical report, Oracle Corporation, August 2005.

[2] A. Berglund, S. Boag, D. Chamberlin, M.F. Fernandez, M. Kay, J. Robie, and J. Siméon. XML path language (XPath) version 2.0. Technical report, World Wide Web Consortium (W3C) Working Draft, 2007.

[3] G. M. Bierman et al. The essence of data access in Cω. In *ECOOP*, pages 287–311, 2005.

[4] S. Boag et al. XQuery 1.0: An XML query language. Technical report, World Wide Web Consortium, January 2007. W3C Recommendation.

[5] D. Chamberlin et al. Programming with XQuery. In *XIME-P 2006*, 2006.

[6] Microsoft Corporation. Open Database Connectivity (ODBC), 2006. http://msdn.microsoft.com/library/en-us/odbc/htm/dasdkodbcoverview.asp.

[7] X-Hive Corporation. X-Hive/DB 7, 2007. http://www.x-hive.com/products/db/index.html.

[8] dom4j Project. dom4j 1.6.1, 2007. http://www.dom4j.org/.

[9] A. Eisenberg and J. Melton. Advancements in sql/xml. *SIGMOD Record*, 33(3):79–86, 2004.

[10] A. Eisenberg and J. Melton. An early look at XQuery API for Java (XQJ). *SIGMOD Record*, 33(2):105–111, 2004.

[11] T. Fiebig et al. Anatomy of a Native XML base management system. *VLDB Journal*, 11(4):292–314, 2003.

[12] Apache Software Foundation. Apache Xindice 1.0, 2007. http://xml.apache.org/xindice/index.html.

[13] M. Harren et al. XJ: facilitating XML processing in Java. In *WWW*, pages 278–287, 2005.

[14] A. Le Hors et al. Document object model (DOM) level 2 core specification. Technical report, WWW Consortium (W3C), 2000.

[15] D. Megginson. SAX: A simple API for XML. Technical report, Megginson Technologies, 2001.

[16] E. Meijer et al. LINQ: reconciling object, relations and XML in the .net framework. In *SIGMOD Conference*, page 706, 2006.

[17] Sun Microsystems. Java database connectivity (JDBC), 2006. http://java.sun.com/javase/technologies/database.jsp.

[18] Sun Microsystems. Java architecture for XML binding (JAXB), 2007. http://java.sun.com/webservices/jaxb/.

[19] C. Mohan and B. G. Lindsay. Efficient commit protocols for the tree of processes model of distributed transactions. In *PODC*, pages 76–88, 1983.

[20] M. Nicola and B. Van der Linden. Native XML support in DB2 universal database. In *VLDB*, pages 1164–1174, 2005.

[21] Oracle. Berkeley DB XML 2.3.10, 2007. http://www.sleepycat.com/.

[22] Castor Project. Castor XML, 2007. http://www.castor.org/.

[23] JDOM Project. JDOM 1.0, 2007. http://www.jdom.org/.

[24] M. Rys. XML and relational database management systems: inside Microsoft SQL Server 2005. In *SIGMOD Conference*, pages 958–962, 2005.

[25] H. Schöning and J. Wäsch. Tamino - an internet database system. In *EDBT*, pages 383–387, 2000.

[26] K. Staken. XML database API draft. Technical report, The XML:DB Initiative, 2001.

[27] J. Suttor et al. JSR 206 Java API for XML processing (JAXP) 1.3. Technical report, Sun Microsystems, September 2004.