ARAXA: an object-relational approach to store active XML documents

Cláudio Ananias Ferraz[1], Vanessa P. Braganholo[2], Marta Mattoso[1]

[1]Programa de Engenharia de Sistemas e Computação - COPPE/UFRJ

[2]Departamento de Ciência da Computação – IM/UFRJ

e-mail: {cferraz, marta}@cos.ufrj.br, braganholo@dcc.ufrj.br

Abstract. *Active XML (AXML) documents combine extensional XML data with intentional data defined through Web service calls. The dynamic properties of these documents pose challenges to both storage and data materialization techniques. We present ARAXA, a non-intrusive approach to store AXML documents. It takes advantage of complex objects from object-relational DBMS to represent both extensional and intentional data. By using a DBMS we benefit from efficient storage tools and query engine. We have defined a storage mechanism with a methodology to materialize AXML documents at query time. We have also implemented a prototype of ARAXA. Our experimental results show that our approach is scalable and extensible.*

1. Introduction

Web Services and XML documents have been intensely used in data exchange and applications communication by using standards. XML has become common sense to data publishing and exchange. Web services, on the other hand, provide simple and non-coupled access to service providers distributed over the Web, which makes application interoperation easier. The success of both of these technologies and the fact that they are well accepted by the industry gave rise to a new class of XML documents: the Active XML (AXML) documents [5]. Active XML documents are basically XML documents composed of XML data (extensional content) together with Web service calls (intentional content). The result of the service calls are embedded within the document.

In the same way as XML documents, AXML documents can be large, which results in problems to manipulate them in main memory. Thus, alternative ways of storing these documents are needed. Queries can also be posed to AXML documents. For this, an XQuery query engine or native XML DBMS could be used. However, the intentional content of AXML documents must be managed during query processing, that is, service calls must be coordinated. The activation of service calls may be associated to query specification criteria, that is, a service may have to be called to answer a given XML query. Service calls may also be completely disassociated from queries. They may need to be made periodically (this periodicity is defined at document design time), independently of query execution time. Due to all of these factors, AXML documents cannot be managed directly with available non-active XML management tools.

Abiteboul et al. [6] developed a platform specifically to manage AXML documents. In [6] AXML documents are stored through file systems, which have several drawbacks (security, indexing, etc.). This solution needs a query system that operates directly on those files, and presents scalability problems. Thus, file system storage is not an alternative when one wants storage and query capabilities in the same solution. A more suitable approach would be to use a DBMS, since it provides both features.

There are several approaches to store and query XML documents in DBMS. Some use Relational DBMS [17], other use native XML storage [15, 29]. However, the active part of AXML documents poses some challenges both to store and query the documents. Relational and native XML DBMS do not support the active feature of such documents. Specifically, they do not know how to deal with the dynamicity of the content, nor with the external data sources (service providers). These features must be considered in a storage system for AXML [13].

Using a native XML DBMS may not be the best alternative to store AXML when companies seek integration with legacy systems. More important, service calls are not directly supported. Relational DBMS support some dynamicity through SQL *triggers*. The dynamicity of triggers could lead one to think on using them to manage web service calls. However, service calls may need to be activated at query time, and triggers can not be activated by SQL SELECT clauses. Thus, they do not have the behavior nor the granularity needed to implement the active characteristic of AXML documents. Consequently, they are not the best alternative to the problem of storing and querying AXML.

Our solution to this problem was found in Object-Relational (OR) DBMS. The modeling of active behavior in OR DBMS is not explicit. However, OR DBMS are capable of dealing with complex objects and associated methods [13]. This allows us to create a class of *active objects* that can be responsible for coordinating service calls and their execution. By using these resources, service calls can be made within SQL queries. It is also possible to create an agent that verifies the periodicity in which a service needs to be called, and can manage these calls automatically. To support XQuery, we can use existing XML-relational storage mappings [12, 18, 19, 32], and consequently, existing algorithms that translate XQuery to SQL queries [18]. This is the direction we take in our approach. We focus on using standard resources in OR DBMS, so that our solution can be applied in any OR DBMS.

This report presents ARAXÁ (A Brazilian city and a Portuguese acronym which loosely translated to English means ***A**n object-**R**elational **A**pproach to store **X**ML Documents with **A**ctive elements*), our proposal to store and query AXML documents. In our solution, we keep the properties of the formal foundation of AXML documents [4]. At the same time, we offer more sophisticated storage resources allied with consolidated query processing capabilities.

The limitation of our approach resides in the mapping between OR and XML. However, our results show a negligible overhead in this transformation. It is, in fact, highly compensated by the fact that an organization can now keep all of their data in a single repository, thus maintenance cost can be reduced, among other benefits such as data integration. Additionally, XML support in these DBMS is always improving. Notice that our solution is DBMS independent. Any OR DBMS can be used.

This report is organized as follows. Section 2 overviews related work and analyzes current solutions for storing and querying AXML documents. Section 3 presents the Active XML Platform developed by the INRIA-GEMO group. Section 4 identifies the difficulties to the problem and proposes a storage schema to AXML documents. Section 5 presents AXML query processing in our storage approach while Section 6 presents the software architecture of ARAXA. Our prototype and experimental results are discussed in Section 7 and 8. Finally, Section 9 concludes this work.

2. Storing Active XML documents

In the literature, the main focus of work related to AXML documents is the development of an initial infrastructure to support execution of service calls and management of results [3-5, 9, 11, 33]. In this way, the problem of storing large amounts of AXML documents has not received much attention.

In the implementation developed in the Active XML Platform by the INRIA-GEMO group [6], active XML documents are stored in a file system directory. This directory is defined by the application, and there is no other storage alternative. Such approach does not provide access control, indexing or data compression. It only can count on services provided by the operational system. In this way, we can anticipate problems with the management of such documents, which can interfere directly in the scalability of the implementation and of the applications that use such documents.

To overcome such problems, the GEMO group proposed the Xyleme-AXML, an implementation of the AXML Peer integrated with the Xyleme Server [34] (a native XML repository). In this approach, the AXML document is stored in Xyleme as if they were regular (non-active) XML documents. The user application must deal with the management of the active part of the documents. Besides, native storage may not be the best alternative to enterprise applications, which store all of their data using relational or object-relational technology. Such data, in a way or another, will likely be related to the AXML documents manipulated by the enterprise. In this case, it would be better to store the AXML documents in the DBMSs already in use in the company. Results on XML storage has shown that object-relational DBMSs are an efficient alternative to store XML[19].

The mapping of XML documents to relational systems have been widely discussed by the database community and several approaches have been proposed [12, 16, 18, 19, 25, 31, 32]. However, the active behavior of AXML brings new aspects involving service calls that are not supported by these alternatives.

There are several approaches to store XML documents. They use from file system to native XML DBMSs [7-9] or Relational DBMSs. Both native DBMSs and Relational DBMSs present good performance when managing XML documents. However, the choice of the storage method must take into account the context and scenario in which the applications that use the documents are in. These properties define the storage strategy to be adopted. The same evaluation must be performed when choosing the storage method to AXML documents.

When we assume a typical commercial scenario, where most of the applications use relational (or object-relational) data, and where years of investments were made to maintain Relational and Object-Relational DBMSs, it is natural to think on a relational alternative to store AXML documents (see Figure 1). The use of a native DBMS, in this case, would represent a considerable extra cost: maintenance of an integration model for DBMSs with different paradigms; acquisition; and training. One of the main advantages of using relational (or object-relational) storage is its maturity and robustness.

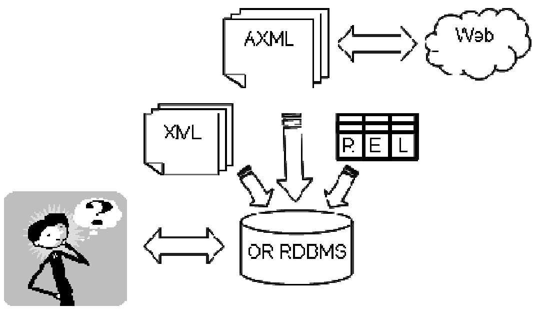


Figure 1. A single data repository in an organization

However, the active property of AXML documents represents an additional complexity in the AXML/Relational mapping. In Relational databases, active features are usually supported by *triggers* [30]. Triggers are a powerful tool to manipulate the dynamic properties of base data. They allow procedures to be automatically started based on the Event-Condition-Action (ECA) paradigm and on temporal aspects. These dynamic behaviors are widely discussed in the Active Database literature and implemented in most Relational and Object-Relational commercial DBMSs. Such behaviors are also discussed for XML documents [10].

Nevertheless, when storing AXML documents into relations we need a class of triggers that is not implemented in most of the commercial DBMSs. This class involves events on selections, that is, triggers started by SQL expressions like "*select <columns> from <tables> where <predicates>*". Such trigger class is needed to activate the service calls when a query is posed in the system.

Another specificity of AXML documents is that a service call may be defined to be executed in a timely manner. A service may need to be called from time to time, or in a specific time. Both native DBMS and Relational DBMS do not provide mechanisms to manage this property. They also do not provide alternatives to embed service call coordination in the DBMS architecture in a non-intrusive and transparent way.

Due to the limitations stated above, our proposal to store AXML documents using the relational model is to use object-relational systems and their complex types. Complex types allow us to associate procedures to a given data type. Such procedures would be able to activate service calls.

3. The Active XML Platform

The Active XML Platform developed by the INRIA-GEMO group [6] is an open-source framework to support Active XML documents in a P2P distributed environment. Figure

X, from [27] shows the file repository for AXML documents and a local query engine to query those files, it also shows the AXML evaluator that interacts with the query engine and the web service execution engine to materialize the documents.

Abiteboul et al. [2] defined the formal model of an AXML document where several materialization strategies can be applied [20, 23]. The materialization of some Active XML data can be either explicitly requested by the user or implicitly triggered by queries that require the (materialized) content of a document.

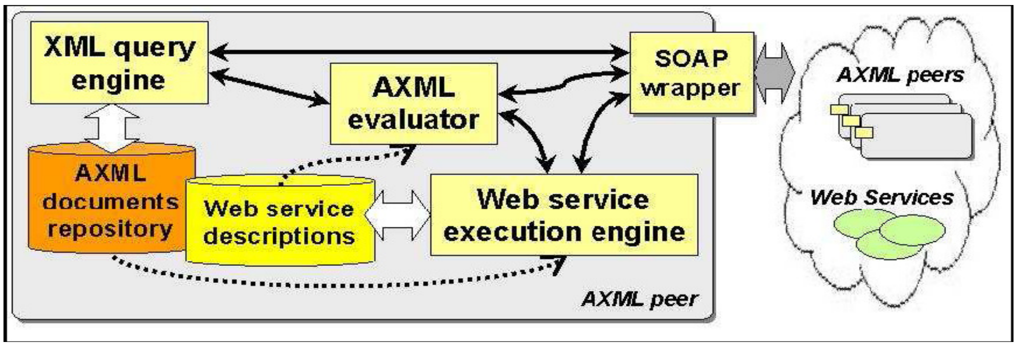


Figure 2 - Active XML Architecture

The internal architecture of an AXML peer, shown in Figure 2, relies on the following modules[6]:

- The **document store**, which provides persistent storage for AXML documents,
- The **evaluator**, whose role is to trigger the services calls embedded inside AXML documents and to update the latters accordingly.
- The **XQuery processor** deals with the service requests, by evaluating the corresponding queries.

Peers communicate with each other *only by the mean of web service invocations*, through their **SOAP wrapper** modules. They can exchange XML data with any web service client/provider, and AXML data with AXML peers.

In this section we describe some characteristics of the materialization process of AXML documents. These specificities were defined by the AXML model [4] Particularly, we discuss their approach in handling the active part of the documents. In section 3.1 we show how services are analyzed for query processing and then in 3.2 some materialization approaches are discussed.

3.1 – Lazy query evaluation

Service calls within a document may have the *Lazy* behavior set by an attribute in the service call. This means that such services must be executed only when needed. Thus, when a query is submitted to a given document, we must analyze the query to minimize the services to be called. More specifically, only services that are essential to a query answer should be called.

Defining the smallest set of services that need to be called to answer a given user query is essential to improve query execution performance. It is clear that the time between a service call and its response may have significant differences from one service to another. Anyway, it must be considered a high cost operation in terms of time, since it is necessary to wait for the remote service provider to return an answer. Thus the number of services to be called must be minimized.

A naive approach would be first to execute the query over the AXML document and ignore the service calls at this point. After processing, the query result would be analyzed and services within it would be called. This idea, however, does not work for two main reasons. First, the result of a query can be large, and finding services within it could require a complete scan over the (large) result, which would be time-consuming. Second, and most important, the query can be formulated over the expected structure of the document (after service call). Queries like */books/book[price > 90]* over the document of Figure 8 would not work in this approach, unless we select all *book* elements and apply the filter over the result after the service materialization. This is not a good approach since it requires an additional query evaluation step.

In [1], Abiteboul et al. present a dynamic algorithm to identify the set of services that must be called to materialize a query answer. The algorithm uses some basic concepts such as: the sequence in which service calls will be made; prune out calls based on their output parameters (contribution to the document) using the WSDL definition of the service; and the use of a service call catalog for fast detection of service calls. These concepts are also adopted in our approach.

Still in [1] the authors present an approach to find the minimal set of services to be executed before submitting the query to the query processor, following the principles of *Linear Path Queries* (LPQ). LPQ is based on the principle that given a query *q* defined by a path expression *p*, a node *n* representing a service call is only relevant to *q* if it is in a path traversed by *p* [1]. Based on this principle, it is possible to generate a set *S* of service nodes that still can contain irrelevant service calls. The service call catalog (which contains the set of services of a given document together with their WSDL definition) can help us generate the set *S*.

In Figure 3 we show an example of LPQ. At the top we show a path expression that retrieves the price of books. The set S is generated by using each step of the path expression concatenated with *(). This represents the service calls.

```
library/books/book/price
          ⬇

/*()
library/*()
library/books/*()
library/books/book/*()
library/books/book/price/*()
```

Figure 3. Example of LPQ

When there are filters in the query expression, it is possible to further prune out irrelevant service calls. However, query filters that involve structures returned by service calls cannot be analyzed at this point (recall the example of */books/book[price > 90]*).

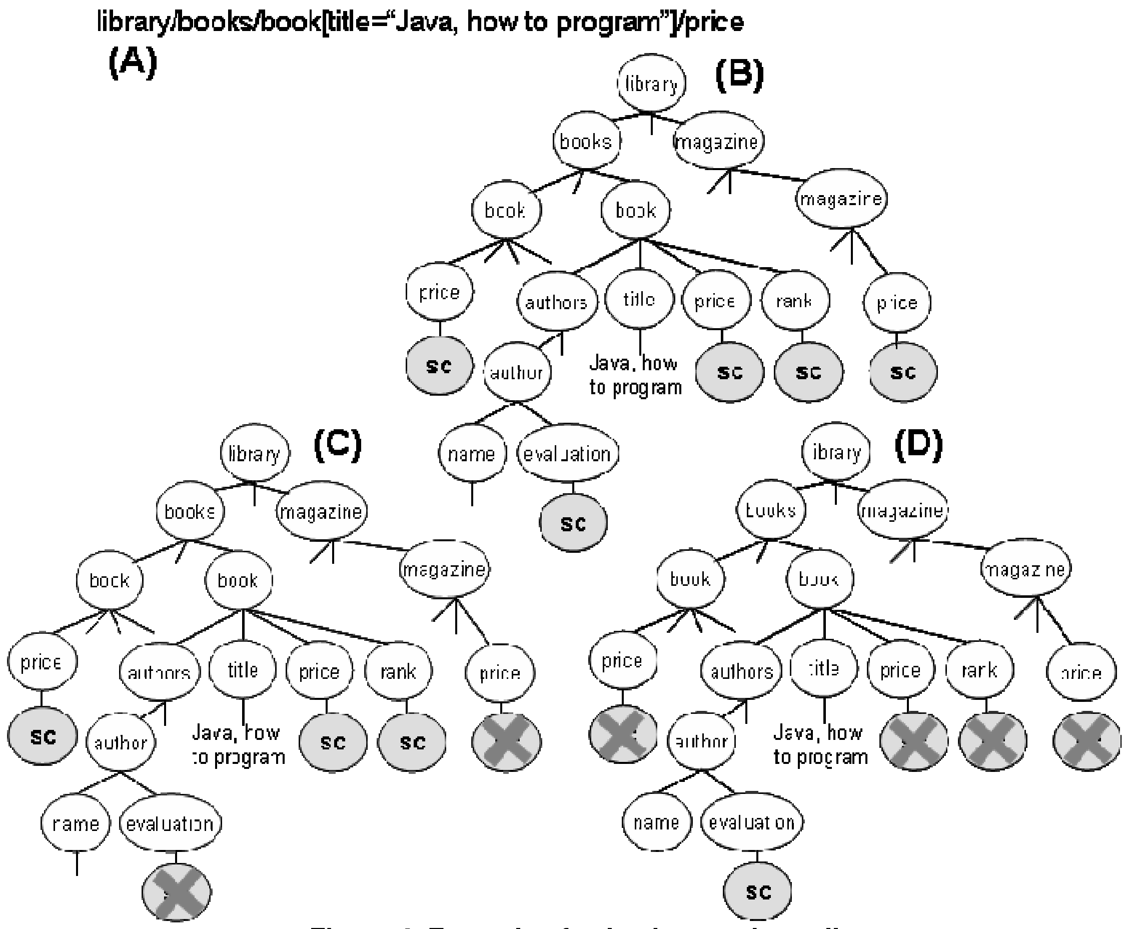


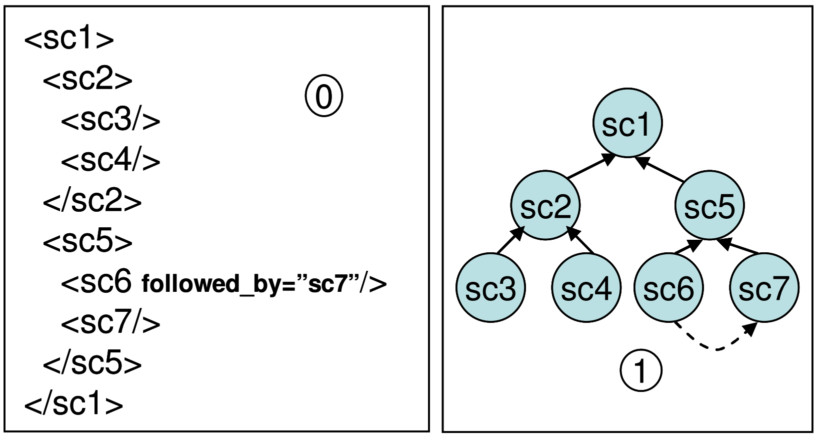
Figure 4. Example of selective service call

Figure 4 shows an example of this selective evaluation. In (A) we show the XPath query that returns the price of the book titled "Java, how to program". In (B) we show a subtree of the AXML document being queried. This subtree shows us that the required information (*price*) is intentional, and must be obtained by a service call. In (C) we highlight the irrelevant service calls for this query that were excluded by using the LPQ principle. In (D) we show the service calls that could be ignored due to the selection criteria of the query, associated with the use of LPQs.

3.2 – Materialization Plans

Once we have the set of services that need to be called (see section 3.1), we need to define an execution plan for this set of services. This is because there may be dependencies between service calls within an AXML document. There can be two types of service call dependencies in an AXML document [27]: dependencies due to nested calls; and dependencies due to the *followedBy* attribute. The *followedBy* attribute allows the AXML document designer to define a sequence in which some services must be executed. Materialization plans must respect this dependency types when it defines the order in which services will be called. In the same way, materialization plans must be efficient. Cost-based optimization for the materialization of AXML documents was first addressed in [28].

The AXML document materialization processes presented in [23] use a service call dependency graph, through a formalism defined in [27] to represent such restrictions. In the graph, each service call is represented as a node. Two nodes n_1 and n_2 are connected if the result of n_2 is required as a (direct or indirect) parameter in n_1. The graphs must have no dependency cycles [27]. An example of dependency graph is shown in Figure 5. In the graph, restrictions due to nesting are represented by continuous arrows, while restrictions defined by *followedBy* are represented by dotted arrows.



```
<sc1>
  <sc2>
    <sc3/>
    <sc4/>
  </sc2>
  <sc5>
    <sc6 followed_by="sc7"/>
    <sc7/>
  </sc5>
</sc1>
```

Figure 5. Example of dependency graph. (0) AXML document; (1) dependency graph of (0) [23]

In this way, we can extract the dependency graph during the mapping process and use it as input to the materialization plan generation proposed in XCraft [27], a cost-based optimizer built upon the Active XML platform. XCraft performs dynamic and

decentralized optimization of materialization plans for AXML documents. Pereira et al. [23] extended XCraft with SLS-MC, an efficient plan generation strategy based on local search and stochastic heuristics. Such a strategy enables XCraft to avoid exhaustive and greedy search methods.

4. Storing AXML Documents in ARAXA

Our approach to store AXML documents uses an Object-Relational DBMS. Through user defined types and methods we create objects to manage remote service calls, as well as an agent that monitors the system clock and verifies the need of calling a given service (those that were defined to be called periodically).

The use of an OR DBMS also keeps the coherence with organizational environments and their needs, since OR DBMS are robust for both storage and querying. In such scenario, a single repository is used to store the company data. This helps the integration of applications that use such data.

Our initial point to map AXML documents to object-relational databases is existing work on XML-relational mapping and on XML-SQL query translation [14, 32]. Among the mapping schemes found in literature, we chose the one proposed by Tatarinov [32]. This approach defines a generic schema to store XML documents which preserves the document order by using a numbering scheme to the nodes. The numbering scheme we use in our work is based on the *dewey* encoding [21]. This encoding minimizes the cost of reordering in cases of updates (insertions and deletions), since only siblings (and their sub-trees) of the updated node must be renumbered. Another important aspect that must be pointed out is the simplicity of the storage schema, which is based on generic structures that are able to represent any XML document, despite its schema. This is an important issue when dealing with documents that have dynamic structures. The result of a service call may be heterogeneous, and using a schema-dependent mapping would not be a good idea (it would need to be frequently modified).

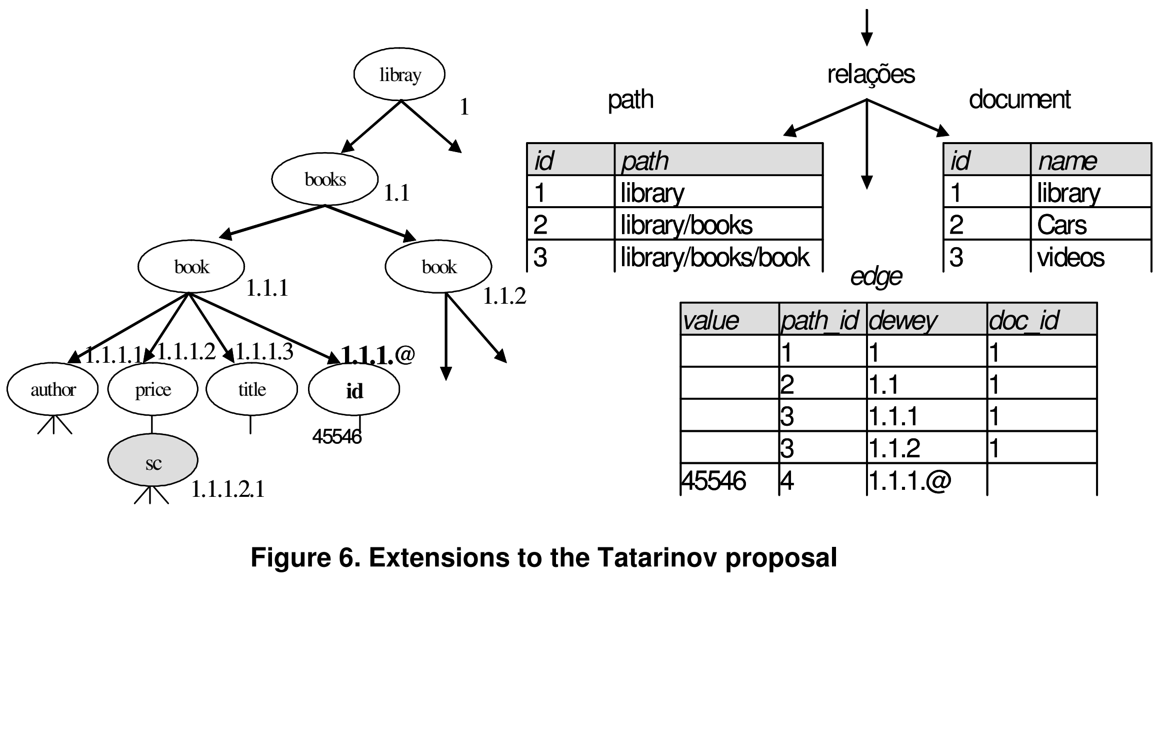
4.1. Mapping XML documents to relations

Two relations are defined in [32] for the mapping, i.e., *Edge (dewey, path_id, value)* and *Path(id, path).* In the *Edge* relation, the attribute *dewey* stores the dewey code of a node. The *Path* relation stores information about the path expressions of the stored elements. This is because generally, the path expression is the same for several different nodes in a given document.

This mapping schema does not distinguish elements from attributes (everything is stored as if it were an element). However, they must be distinguished somehow if we want to reconstruct the stored document. Besides, this schema does not support the storage of several documents – the approach deals with only a single document.

To overcome these limitations, we propose two extensions. The first one addresses the storage of attributes. Here, we benefit from the fact that there is no order between attributes within an element. Thus, we propose to store attributes using the same *dewey* code of its parent element. To differentiate an attribute, we also add the "@" symbol to the attribute name. In this way, it is possible to reconstruct the stored document exactly as it was before being stored. The second extension we propose is the addition of a new relation in the mapping schema, i.e., *Document (id, doc_name).*

Also, we add a *doc_id* column to the *Edge* relation and make it a foreign key to the *Document* relation. This extension solves the problem of storing several documents. The extensions we describe here can be seen in Figure 6.



id	path
1	library
2	library/books
3	library/books/book

id	name
1	library
2	Cars
3	videos

value	path_id	dewey	doc_id
	1	1	1
	2	1.1	1
	3	1.1.1	1
	3	1.1.2	1
45546	4	1.1.1.@	

Figure 6. Extensions to the Tatarinov proposal

Our mapping benefits the selective evaluation of service calls when there are filters in the query expression. This is possible by joining the *edge* and *path* relations, searching for node-value combinations that match the filter. Notice here that, the joining of the edge and value tables is usually faster than calling lots of irrelevant services and waiting for them to return an answer.

This mapping schema also favors LPQ pre-analysis, since the *path* relation represents a compacted document structure (it has all path expressions of the document, with no duplicates). The *path* relation helps us to match LPQ path expressions with path expressions that contain service calls within the document.

4.2. Representing Web services in object-relations

Once the mapping has been defined, the dynamic properties of the documents must be managed. The active behavior of the document must be taken into account not only at storage time, but also at query execution time. As mentioned before, service calls embedded in the document are responsible for the active part of a document. These calls must then be identified. This is relatively easy once the names of elements that represent service calls are standardized: <sc>. We store this information in a Service Call Catalog. The Catalog is represented by an additional schema in the DBMS, and it has the following structure:

Service_call(id, path_id, dewey, doc_id, serviceURL, methodName, serviceNameSpace, useWSDLDefinition, signature, callalbe, frequency, lastcalled, followed, mode, doNesting)

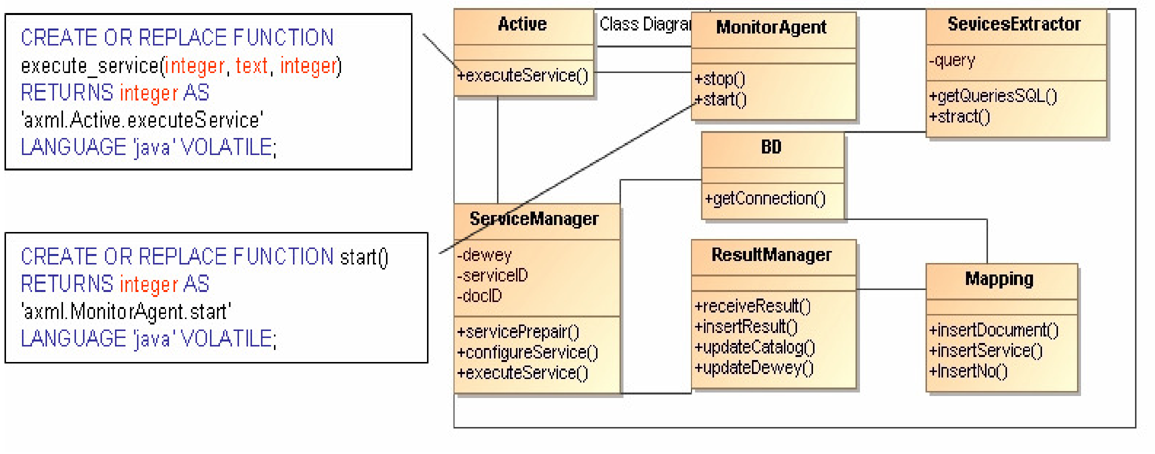
Parameter(id, service_id, path_id, type, name)

The *Service_call* relation stores all service calls within a given AXML document. It also stores the information of which document they appear, and where they are located within the document. This relation also stores the service call attributes, according to the AXML model. The *Parameter* relation stores the parameters that will be passed to the service provider during the execution of a service call.

4.3. Managing Web Service calls in object-relations

Once AXML documents are stored, we need an infrastructure that is able to call services when needed. These calls are needed both at query time (a service call may return results needed to answer a given query), and at a specific time (for services that require time-based executions). In this section, we show such infrastructure.

In our approach, services are called through an SQL query of type *select execute_service(doc_id, service_id, dewey*. In this select clause, *execute_service()* is a method call. This method, added by our mapping strategy, is a generic client method for Web Services.



```
CREATE OR REPLACE FUNCTION
execute_service(integer, text, integer)
RETURNS integer AS
'axml.Active.executeService'
LANGUAGE 'java' VOLATILE;
```

```
CREATE OR REPLACE FUNCTION start()
RETURNS integer AS
'axml.MonitorAgent.start'
LANGUAGE 'java' VOLATILE;
```

Figure 7- Class diagram of the AXML storage components

This generic Web Service client is implemented independently of the DBMS. It is associated with the OR-DBMS simply by associating a function defined in the database schema with a method of the Web Service client. This association process is available in most of the OR-DBMS, since they support high-level programming languages.

In Figure 7 we present a simplified class diagram of our infrastructure together with an example of how this infrastructure is associated to the DBMS. The components were developed in Java, outside of the DBMS. We used SQL UDF functions and associated them with our implemented classes. As an example, we have defined a UDF function in SQL called *start()* (lower-left box in Figure 7), and associated it with the *start()* method of the *MonitorAgent* class. When we call the *start()* function in the DBMS, the *start()* method of the *MonitorAgent* class is called. It is responsible for initiating the agent that monitors the system clock and verifies the need of calling a given service that has its execution based on time events. The other UDF SQL function shown in Figure 7, *execute_service(),* is connected to the *executeService()* method of class *Active*. It represents a generic Web Service client that is able of calling services, and also to delegate to other components in our approach important tasks related to materialization and mapping of results. In Figure 7, the functions use the syntax of PostgreSQL, however, similar mechanism are available at Oracle 9i [22], IBM DB2 [7], among others.

It is important to notice that the implementation was developed outside the DBMS and bounded with the DBMS later on through functions that associate the high-level language module and the database schema. We provide details of the implementation in Section 7. Clearly, this mechanism does not interfere in the internal structure of the DBMS (it does not need to be altered or recompiled). In this way, the same Web Service client implementation can be used in different DBMSs.

5. AXML materialization during query processing in ARAXA

Executing a query on AXML documents may involve materializing active elements. There are several alternatives to execute the service calls of the materialization. Optimization strategies have been proposed for such materialization while preserving the document properties [1, 23]. In [1] different alternatives are proposed to avoid materializing elements that will not take part on the query evaluation. Thus, they present algorithms to identify just the services that have to be executed. Ruberg [23] show how to extract the dependencies on these service executions and present a dependency graph generator. Based on this graph Pereira et al. [23] propose optimization strategies for these service executions.

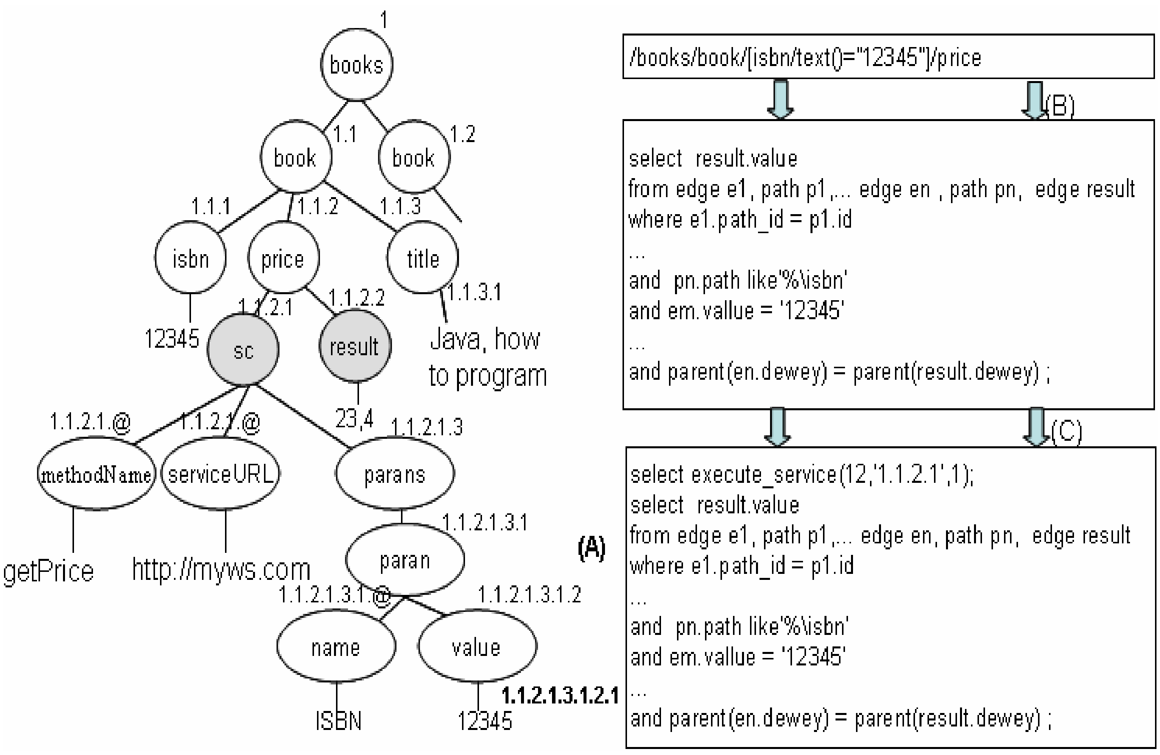
In ARAXA, to process queries over AXML documents we take advantage of those previous successful techniques by adapting them to our storage structure. Our methodology involves the following steps: (i) identify services that need to be called to answer the query; (ii) translate the query from XQuery to SQL; (iii) identify the dependencies among service calls; (iv) define the order and call the services; (v) store service call results in the relational tables using the same mapping that was used to store the document; (vi) execute the query; (vii) map the resulting tuples to XML and return the answer to the user. We explain each of these steps below.

In step(i), we must analyze what services are relevant to answer that query. This is because depending on the query, the result of a service call may not contribute to the final answer. We have used the *lazy query evaluation* mechanism from [1] to the identification (see details on Section 3).

Step (ii) translates the XQuery/XPath query to SQL and during step (iii) we use the dependency graph generator from [27] to rewrite this SQL query to include queries that will actually call the services by using the *execute_service()* function. To define the execution alternatives for the services that will be actually invoked we have adapted the SLS approach [23] to the possibilities in the *execute_service()* function. In [27] the evaluation of materialization plans occurs with delegation of control of service execution in a peer-to-peer network, with a Master Site orchestrating only the initial execution. In the step (iv) of our approach, we use this algorithm only to define an optimized execution order to the service calls and do not delegate service execution control. Thus, we assume the host DBMS represents the Master and the orchestrating site, and all the executions are controlled by this same site.

To better understand these four steps, take a look at Figure 8**.** In (A) we show a sub-tree of an AXML document that has a service call; in (B) we show the XPath $\rightarrow$ SQL query translation process; in (C) we add the queries needed to activate the service call.

In the example, the document contains information about books. Each book has author, price, ISBN, etc. The price information is dynamic, and it is provided by a service call. The ISBN of the book is passed to the service as a parameter. When the user submits a query that contains the book price in the result, the price information needs to be materialized (that is, the call to the price service needs to be executed).



```
/books/book/[isbn/text()="12345"]/price
```

```
select  result.value
from edge e1, path p1,... edge en , path pn,  edge result
where e1.path_id = p1.id
...
and  pn.path like'%\isbn'
and em.vallue = '12345'
...
and parent(en.dewey) = parent(result.dewey) ;
```

```
select execute_service(12,'1.1.2.1',1);
select  result.value
from edge e1, path p1,... edge en, path pn,  edge result
where e1.path_id = p1.id
...
and  pn.path like'%\isbn'
and em.vallue = '12345'
...
and parent(en.dewey) = parent(result.dewey) ;
```

Figure 8 - Example of the query translation mechanism

After query translation, we add the *execute_service()* calls to the SQL (as shown in (C)) and execute the final query. In our approach, the order in which services will be called is imposed in the moment we build the queries that will activate the service calls (item (c) of Figure 8). Service calls that have any execution order restriction are defined in distinct queries, respecting the order in which they must be executed:

select execute_service(…);
select execute_service(…);

On the other hand, service calls that can be executed in parallel are defined in a same query: *select execute_service(...), execute_service(...);* .

The parameters needed for the service call are taken from the catalog (see more details in Section 6). Notice that, as a result, we have a set of SQL queries. In step (v), results of each service execution are inserted in the stored AXML document using the mapping rules. The results are embedded in a <*result*> element, which is inserted as the immediate right sibling of the <*sc*> element. This is why *execute_service()* statements are not sub-queries – this is not necessary, since they modify the database state. The Service Call Catalog is also updated (it contains information such as time of last execution of a given service, among others).

Then, the step (vi), SQL query execution, can benefit from the DBMS query engine. After this execution, in step (vii) the obtained (relational) result needs to be mapped to XML (as it is expected as a result of an XPath/XQuery query). This result construction is based on the XPath/XQuery query structure. This is a post-processing step of our approach, and its goal is to make our storage proposal completely transparent to the user.

6. Architecture of ARAXA

In the preceding sections we proposed a solution to the problem of storing and querying AXML documents using an OR-DBMS. In this section we present the architecture of our approach. The architecture shown in Figure 9 is composed of two main modules: the *Control Module* and the *Integration Module*. They are further divided into sub-modules.

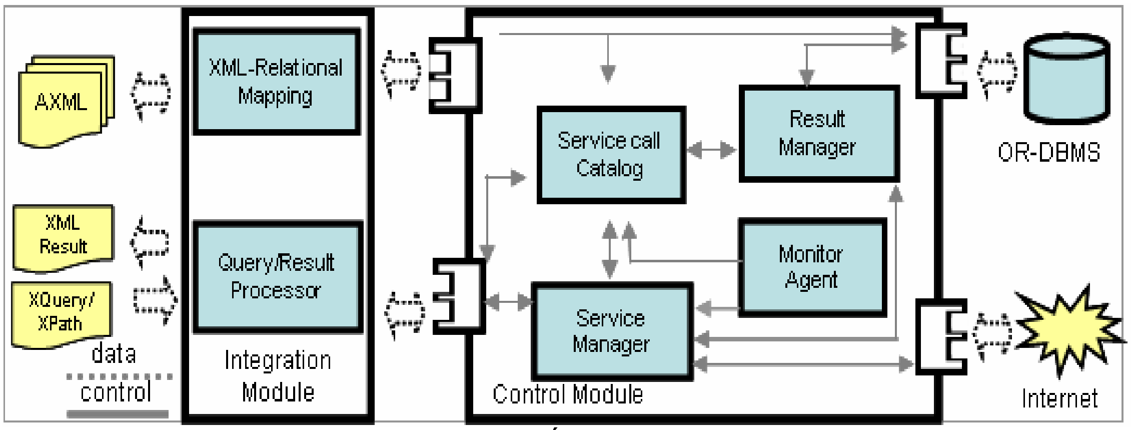


Figure 9. ARAXÁ Architecture

The Integration Module is composed of a *Query Translator*, and an *XML-Relational Mapper*. The Integration Module is not related to the DBMS, that is, it executes externally to the DBMS, acting as a client application. The XML-Relational Mapper receives an AXML document and stores it in the relations defined by our mapping schema (see Section 4). During this process it identifies the active parts of the document and stores this information of the Service Call Catalog of the Control Module.

During this mapping we also generate the service calls dependency graph of the document. Information contained in the graph and in the Service Call Catalog is used in the execution of service calls.

The Query Translator module translates an XQuery/XPath to SQL and identifies the services that need to be called. It is also responsible for mapping the relational query result back to XML. The algorithm we use in this module to translate XML to SQL queries is based on the high-level algorithm proposed by Tatarinov [31].

The Control Module is responsible for maintaining the system transparent to the final user. It is composed of the *Service Call Catalog*, *Service Manager*, *Results Manager* and *Scheduler Agent*.

The Service Call Catalog stores information about sub-trees that represent service calls. This information includes: behavior defined by the designer; service call criteria; activation parameters; service call location within the document; statistics about service execution and service providers; and service calls dependency graph. Initially, the Catalog is populated with information extracted during the XML-Relational mapping. The Catalog provides information to the other architecture components, acting as a guide to queries and decision-making. However, it does not perform any activity in the system. It is simply a data source that is fed and queried by the other architecture components.

The Service Manager represents a generic Web Service client. It is activated by the *execute_service()* procedure. This component accesses the Service Catalog to find the parameters that must be used in the service call. It also verifies if the service call really needs to be made. If so, it calls the service by communicating with the external environment, and then passes the obtained answer to the Results Manager.

The Results Manager is responsible for materializing the result of a service call within the mapped AXML document. It applies to the resulting XML tree the same XML-Relational mapping used to store the original document. The behavior that the materialization should follow is taken from the Service Call Catalog. The materialization behavior is given by the *mode* attribute in the service call definition. This attribute value defines two distinct behaviors: *replace* - replace the old XML forest by the new one; *append* - append the new XML forest next to the previous one. This is the default behavior. The Service Call Catalog is then updated after the service execution (last time the service was called, time of response, etc.).

The Scheduler Agent executes service calls that were defined by the designer to be executed within a given time interval or specific date. This behavior can be set on a service by using the *frequency* attribute. The possible values for this parameter are: *Once* – the service is executed only once, at system start time; *Lazy* – the call is only executed when its results are needed; *On Date* – the service should be executed at a specific data/time (for instance *frequency*="12/25/07 14:36"); *Every X* – the service will be called every *X* milliseconds (example: *frequency*= *"every 60000"*). The agent continually monitors the system verifying the need to call services. When it identifies a service that needs to be called, it delegates the service activation to the Service Manager.

Our architecture integrates naturally with legacy systems. It also has the advantage of using efficient XML-Relational mapping techniques [14]. Furthermore, it can be implemented in most of open-source and commercial DBMSs.

7. Prototype

A first prototype of our architecture has been developed. We used PostgreSQL [26] as our Object-Relational DBMS. The Control Module was implemented internally to the DBMS using PLJava [24] together with APIs for Java, Web Services and XML. The use of PLJava allows a loose coupling between the implementation and the chosen DBMS. This is because the implementation can be developed independently, and then associated with the DBMS through the function association mechanism. This mechanism allows us to associate a set of Java classes with a schema within the DBMS.

The Integration Module was developed in Java. The query translation component is under development based on the algorithms proposed in [31]. Figure 10 shows the technologies we have used in the prototype, as well as how they are organized in the implementation.

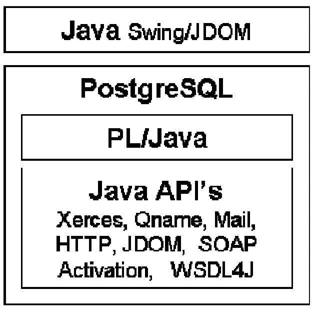


Figure 10. Technologies used in the prototype

8. Experimental Evaluation

In this section we present some results obtained from experiments with the ARAXA prototype. The main goal of our experimental evaluation is to analyze the coupling of active data management strategies in OR-DBMS. We also aim at evaluating the impact of mapping XML to OR in the context of AXML querying and materialization.

The tests were executed in an Intel Pentium 4 2.66 GHz, 480 MB RAM machine running Windows XP. The DBMS was PostgreSQL 8.2. We also used the JRE 1.5.0_11 Java virtual machine. To generate the AXML documents, we have used ToXgene [8]. In our experiments, we explore four scenarios to execute queries and evaluate the performance of each of them in every step of the execution, according to our methodology (see Section 5).

The documents we used in each scenario represent a book collection. Each sub-tree contains several information about a given book. One of them is the book price, which is an active element and defined by a service call. The service call uses the book ISBN as input parameter. The documents were generated with four different configurations. Figure 11 shows details about the documents. Document 1 has, besides books, a collection of magazines (2 books and 13 magazines). The prices of the magazines are also provided by a web service call. Despite having the same number of service calls (much higher than document 1), documents 2 and 3 (500 books each) are different because document 3 has more information on each book sub-tree (abstract and reviews). Document 4, despite having the same number of nodes of documents 2 and 3, has nodes with more data volume, which increases its size on disk significantly.

These scenarios were chosen to evaluate several aspects: Document 1 is small and has few service calls. It can easily be handled in main memory. Document 2 is also small, but has 500 service calls, which we believe is hard to manage in main memory only. Document 3 has the same number of service calls, but is larger. However, it may still fit in main memory. Document 4 aims at representing documents that do not fit in memory, and could not be processed by file system based platforms such as AXML [6] .

In all of the documents we performed a query to retrieve the price of a given book (*/catalog/book[isbn="6952254279"]/price*). To execute this query, we evaluated two strategies to detect relevant services, LPQ and Filters, detailed in Section 3.1 (both are implemented in the prototype).

Since our XPath/SQL translation algorithm is not fully implemented yet, we generated the SQL query corresponding to the XPath query by hand (the query corresponding to item B of Figure 8), and then the implementation included all the service calls (*select execute_service*) as needed. All the remaining steps of our methodology were executed automatically by the prototype.

Figure 11 show the results for each strategy in all scenarios. For each case we have executed the query 10 times. The results we present here are the means of these executions. In the graphics, we show the complete time of the execution discriminating the time spent on specific tasks according to our methodology (Section 5) as follows:

1. *Services extraction* – step (i) of our methodology (identify the services that need to be called). Each figure uses a different strategy at this point (filters or LPQs)
2. *Services parameterization* – this step is performed in step (iv) of our methodology. Basically, this comprehends taking all the information needed to call a given service from the service call catalog and from the AXML document, and then generate the SOAP message
3. *Services execution*– this step is not part of our methodology. It expresses the time spent on the messages exchange and the remote execution of the service.
4. *Materialization* – step (v) of our methodology (store service call results in the relational tables using the same mapping that was used to store the document).
5. *Final Result* – step (vi) of our methodology (execute the SQL query).

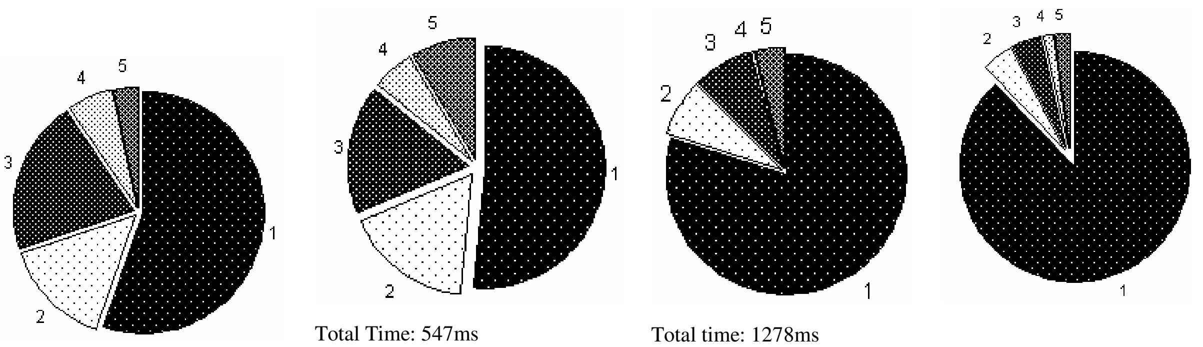
Steps 1 to 3 are specific to the materialization of the active part of AXML. Those steps are present in any AXML system. In fact, we have adapted previous solutions from [6]. Step 4 shows the overhead of our methodology through the mapping process. We can see that this time step is negligible to the rest of the steps. Then, Step 5 is also present to other AXML solutions but in ARAXA it can take benefit of the local DBMS query execution engine. Notice that we could not measure the complete overhead of our methodology, since we do not have all the steps implemented yet. Performance results from previous work in XML-OR mappings [17] suggest that they are also not relevant considering the time spent to handle the active part of the document.

Document 1
8 Kb, 15 Service Calls, Tuples in edge relation: 284

Document 2
293 Kb, 500 Service Calls, Tuples in edge relation: 13115

Document 3
11,5 Mb, 500 Service Calls, Tuples in edge relation: 252976

Document 4
500 Mb, 500 Service Calls, Tuples in edge relation: 252976



Total Time: 547ms

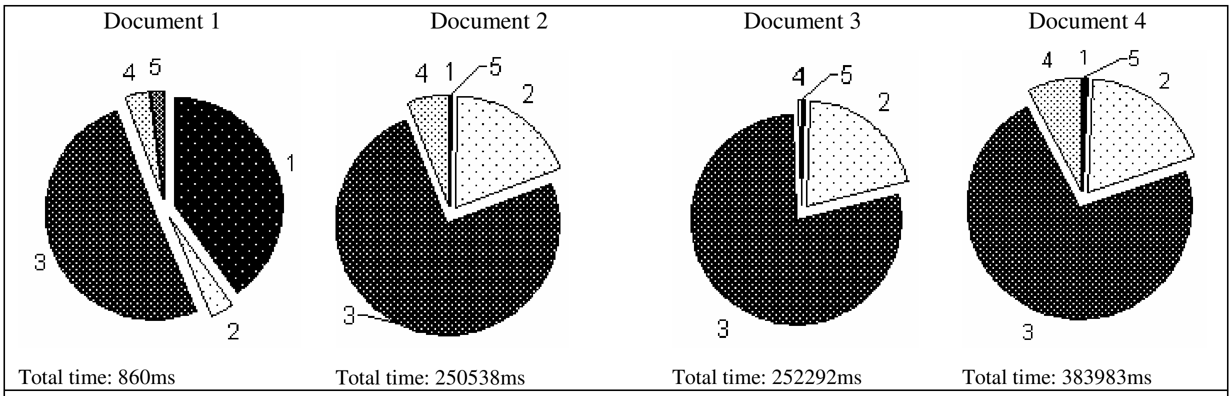
Total time: 1278ms

Total Time: 502ms

Total time: 2391ms

Steps of the query processing:
1- services extraction 2- services parameterization 3-Services execution at remote sites 4- materialization 5- final result

Figure 11. Experimental results using filters



Total time: 860ms Total time: 250538ms Total time: 252292ms Total time: 383983ms

Steps of the query processing methodology:
1- services extraction 2- services parameterization 3- Services execution at remote sites 4- materialization 5- final result

Figure 12. Experimental results using LPQ

The results we obtained show that the filters strategy performs much better than LPQ on our scenarios. The filters strategy has reduced the number of services to be called to the lowest possible limit (which is 1 in this case), while the LPQ strategy resulted in all 500 service calls in documents 2, 3 and 4. It is important to notice, however, that the structure of the AXML documents 2, 3, 4 that we used in our experiments does not benefit from LPQ, since our document has a regular structure with service calls regularly distributed over the document. In this way, when we execute a regular linear query such as */catalog/book/isbn/price* over the document, all service calls within this path are retrieved i.e., no service call was discarded. However, the LPQ strategy eliminated 92% of the irrelevant service calls in document 1. This is because the document structure is irregular (books and magazines), thus the path expressions could cut out service calls that retrieve magazine prices.

Our results also show the importance of discarding as much irrelevant services as possible. In Figure 12, step 3 (services execution in remote sites) took a long time, since lots of services were called. In Figure 11 this wait time is smaller. In Figure 11, however, the largest time was dedicated to service parameterization. This points out to the need of optimizations in this step.

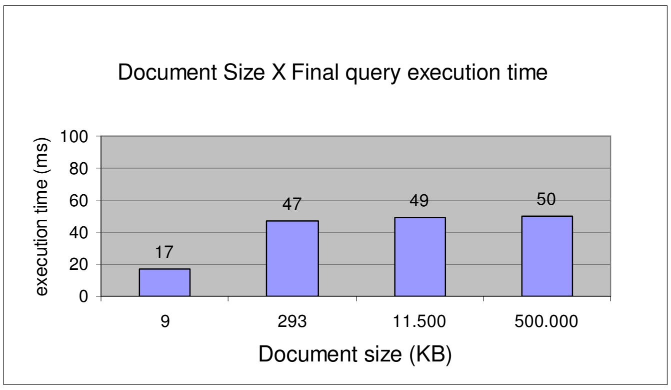


Figure 13- Final query execution time

We can also observe, by this experiment, the scalability of our implementation. The performance on query execution reduces in a much smaller proportion than the increase in document size. We can also notice that the SQL query execution time (the one corresponding to the XPath/SQL translation) performed satisfactorily, since it had no significant time variation on the smaller and bigger document (47ms and 49ms respectively). This behavior is shown in the graphic of Figure 13. In this graphic, we show the relation between document size and the time of the last step of the query execution process (execution of the SQL query translated from XQuery) over the AXML document. At this step, all service calls were already materialized. Notice that there was a very small variation on the query execution time, even when the document size increases significantly.

Another criterion we analyzed in our experiment was the Monitor Agent. We have made tests both with the Monitor on and off. In Figure 14, we show the behavior of memory usage of the PostgreSQL DBMS we used in our evaluations. In the Figure, we show a sum of the memory usage of all process related to the DBMS. This was observed in an interval of 240 seconds. At second 95, we started the Scheduler Agent. By looking at Figure 14, we can observe that, even after the Agent startup, there was no change in memory usage on the DBMS.

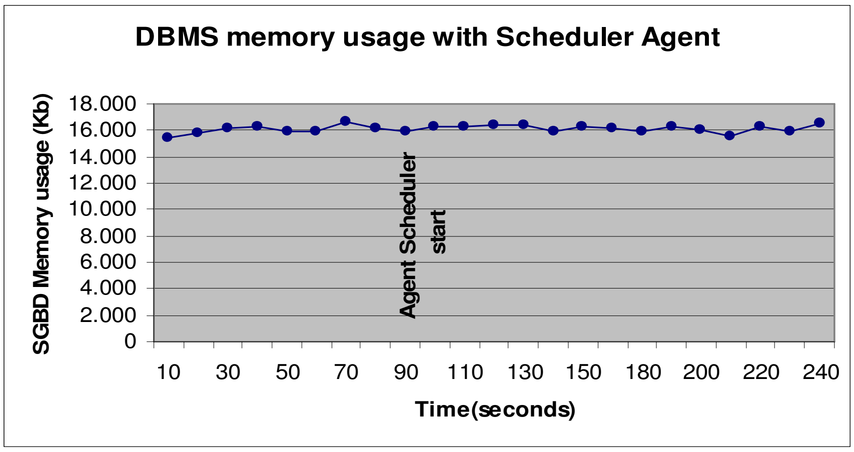


Figure 14 - DBMS memory usage with Scheduler Agent

Two steps proved critical in terms of performance: the parameterization of the service call and the service executions. These results, however, were not influenced by our approach. The wait-time for results depends on external factors (such as bandwidth and service provider), even when we optimize the orders in which the services need to be called and use low-cost equivalent services [23]. The service parameterization, on the other hand, can be optimized, for instance, by analyzing the signature of parameters in the WSDL, using cache, and retrieving the parameters from past service calls.

It is important to state that our approach can be extended to include other algorithms to eliminate irrelevant service calls. We initially implemented the two we analyze here (LPQ and filters), but others can be added with no difficulty.

The results we obtained show several important points: (i) the relevance of the optimizer in the performance of materialization and query execution, especially in tasks related to service extraction; (ii) our approach can be used in conjunction with the OR-DBMS query optimizer; and (iii) several strategies of service management can be plugged into our architecture.

As a summary, the prototype has proved itself scalable and non-intrusive. It has shown that the critical execution time is concentrated on handling the services, independent from the storage structure. In addition, the 500Mb document (scenario 4) did not fit in main memory on our tests, and thus it is not likely to be handled by the available implementation of the AXML platform [6] .

9. Final Remarks

In this report, we present ARAXA, a solution to the storage and querying of Active XML documents. Our approach takes advantage of query processing and complex object representation of the Object-Relational paradigm. The OR paradigm is robust and used in most commercial applications. To map AXML to Relations, we use and extend the efficient approach from [32]. Specifically, we propose an extension to map attributes and to store multiple documents. We have also defined a query processing with materialization methodology into the ARAXA architecture and a prototype that (partially) implements this architecture.

Our experimental results points to the need of optimization strategies to selectively evaluate the services in the AXML documents when answering queries. As future work, we plan to investigate different selective evaluation techniques and a combination of several techniques to be applied according to the document structure.

Our proposed solution to the storage of AXML documents keeps the properties of the AXML model. This can be stated since we use well known algorithms that have already been proved correct in literature: the algorithm of Tatarinov to store the documents; materialization algorithms of the AXML model (LPQ, catalog and filters); and Ruberg's algorithm to preserve the dependency of the service calls.

We want to emphasize, however, the benefits of integration with legacy systems. The mapping schema we propose is not specific to AXML documents, and so it is also capable of storing and querying regular XML documents. Additionally, we have experimentally observed that the mappings imposed by our approach are negligible to the materialization process and it does not interfere in the DBMS performance (see the memory graphic in Figure 14). Thus it does not affect any legacy application currently running on the DBMS. Another benefit of our approach is that OR and XML data can be stored in a single repository.

Even though we have focused our solution on AXML documents, the combination of XML extensional data with Web services is expected to be present in several Web documents, independent of an explicit platform such as Active XML to handle them. Thus, our architecture can be seen as an alternative integration model between data and services, since it allows a new, simple and non-coupled way of integrating data through the use of Web Services, which can be activated by methods in OR DBMSs. This model, as well as the Active Database Model, brings a new dimension of dynamic properties, which is essential to modern computational environments.

References

1. Abiteboul, S., et al. *Lazy Query Evaluation for Active XML*. In *SIGMOD*. 2004. France.
2. Abiteboul, S., et al., *Web Dynamics: Adapting to Change in Content, Size, Topology and Use*. 2004: Springer. 275-300.
3. Abiteboul, S., Benjelloun, O. and Milo, T., *Active XML Primer* in *GEMO Report number* 2003, INRIA Futurs.Technical Report 275. Available from: ftp://ftp.inria.fr/INRIA/Projects/gemo/gemo/GemoReport-307.pdf.
4. Abiteboul, S., Benjelloun, O. and Milo, T., *The Active XML project: an overview*. 2005, GEMO INRIA.Technical Report 331.
5. Abiteboul, S., Benjelloun, O. and Milo, T. *Positive Active XML*. In *PODS* 2004. France.
6. ActiveXML. *ACTIVE XML WEBSITE*. 2007; Available from: http://activexml.net/.
7. Almeida, M.S., et al., *DB2 Java Stored Procedures*. 1ª ed. IBM Red Books, ed. I.I.T.S. Organization. 2000. 418.
8. Barbosa, D., et al., *ToXgene: An extensible template-based data generator for XML*. SIGMOD, 2002.
9. Benjelloun, O., *Active XML : A data-centric perspective on Web Services*, Doutorado, lÚniversité Paris XI (10/2004),2004
10. Bonifati, A., Ceri, S. and Paraboschi, S., *Active rules for XML: A new paradigm for E-services*. The VLDB Journal — The International Journal on Very Large Data Bases, 2001. **10**(1): p. 39--47.
11. Canaud, E., Benbernou, S. and Hacid, M. *Managing trust in Active XML, Services Computing*. In *IEEE International Conference on Publication*. 2004.
12. DeHaan, D., et al. *A Comprehensive XQuery to SQL Translation Using Dynamic Interval Encoding*. In *SIGMOD*. 2003. USA.
13. Ferraz, C., Braganholo, V. and Mattoso, M. *Uma Abordagem para o Armazenamento de Documentos XML Ativos*. In *WTDBD*. 2006. Brazil, In Portuguese.
14. Florescu, D. and Kossman, D., *Storing and Querying XML Data using an RDBMS*. IEEE Data Engineering Bulletin, 1999. **22**: p. 27-34.
15. Jagadish, H.V., et al., *TIMBER: A native XML database*. VLDB Journal, 2002. **11**(4): p. 274-291.
16. Vieira, Humberto, *Xverter: Armazenamento e Consulta De Dados XML Em SGBDs*, Msc. Thesis, COPPE/UFRJ,2002
17. Khan, L. and Rao, Y. *A Performance Evaluation of Storing XML Data in Relational Database Management Systems*. In *WIDM*. 2001. USA.

18. Krishnamurthy, R., Kaushik, R. and Naughton, J.F. *XML-to-SQL Query Translation Literature: The State of the Art and Open Problems*. In *XSym*. 2003. Germany.

19. Leonard, J.L., *Strategies for Encoding XML Documents in Relational Databases: Comparisons and Contrasts*, Master Thesis, East Tennessee State University.,2006

20. Milo, T., et al., *Exchanging intensional XML data.* ACM Transactions on Database Systems (TODS), 2005. **30**(1): p. 1-40.

21. OCLC. *Introduction to the Dewey Decimal Classification*. ONLINE COMPUTER LIBRARY CENTER 2002; Available from: http://www.oclc.org/oclc/fp/about/about_the_ddc.htm.

22. Oracle. *Oracle 9i*. 2007; Available from: http://www.oracle.com/technology/sample_code/tech/java/jsp/oracle9ijsp.html.

23. Pereira, D., Ruberg, G. and Mattoso, M. *Geração Eficiente de Planos de Materialização para Documentos XML Ativos*. In *SBBD*. 2006. Brazil, In Portuguese.

24. PLjava. *The pljava Project* 2006; Available from: http://gborg.postgresql.org/project/pljava/projdisplay.php.

25. Plougman, D., Lauritsen, T. and Olesen, J. *Shredding and Querying XML Data Using an RDBMS*. Eletronic Document Libray 2004; Available from: http://www.cs.aau.dk/library/files/rapbibfiles1/1085738327.ps.

26. Postgresql. *Postgresql*. 2007; Available from: http://www.postgresql.org/.

27. Ruberg, G. and Mattoso, M., *XCraft: A Dynamic Optimizer for the Materialization of Active XML Documents*. 2006, COPPE/UFRJ: Rio de Janeiro.Technical Report Reviewing process. Available from: http://www.cos.ufrj.br/~gruberg/xcraft_rt.pdf.

28. Ruberg, N., Ruberg, G. and Monalescu, I., *Towards cost-based optimization for data-intensive Web service computations.* Proceedings of SBBD, 2004: p. 283-297.

29. Schöning, H. *Tamino - A DBMS designed for XML*. In *ICDE*. 2001.

30. Tanaka, A.K. *Bancos de Dados Ativos*. In *SBBD*. 1995. Brazil.

31. Tatarinov, I., *Storing and Querying Ordered XML using a Relational DBMS*. 2002, Tech Report Univ. of Washington: Washington. p. 13.Technical Report.

32. Tatarinov, I., et al. *Storing and Querying Ordered XML Using a Relational Database System*. In *SIGMOD*. 2002. USA.

33. Vieira, H., Ruberg, G. and Mattoso, M. *XVerter: querying XML data with OR-DBMS*. In *WIDM*. 2003.

34. Xyleme. *XYLENE WEBSITE.* 2006; Available from: http://www.xyleme.com/.