ARAXA: storing and managing Active XML documents

Cláudio Ananias Ferraz¹, Vanessa Braganholo², Marta Mattoso³

¹Computer Science Department, COPPE/Federal University of Rio de Janeiro, Brazil
{cferraz, marta}@cos.ufrj.br

²Computer Science Department– IM/Federal University of Rio de Janeiro, Brazil
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. In this paper, we present ARAXA, a non-intrusive approach to store and manage AXML documents. We also define a methodology to materialize AXML documents at query time. The storage approach of ARAXA is based on plain relational tables and user-defined functions of object-relational DBMS to trigger the service calls. By using a DBMS we benefit from efficient storage tools and query optimization. Approaches without DBMS support have to process XML in main memory or provide for virtual memory solutions. One of the main advantages of ARAXA is that AXML documents do not need to be loaded into main memory at query processing time. This is crucial when dealing with large documents. The experimental results with ARAXA prototype show that our approach is scalable and capable of dealing with large AXML documents.

1. Introduction

Several aspects in real world nowadays are dynamic: dynamic web pages, dynamic systems, dynamic databases, etc. In this dynamic world, interoperability is crucial. Web Services provide simple and non-coupled access to service providers distributed over the Web, which makes application interoperation easier. On the other hand, XML documents have also been used to application interoperability. In this scenario, it is natural to think on dynamic documents. Such documents combine extensional content with intentional content, which is obtained through Web Service calls. Abiteboul et. al [2] developed a framework to manipulate active XML (AXML) documents. In their framework, the results of the service calls are embedded within the XML document. Figure 1 shows an example of an AXML document. As defined in [2], the <sc> nodes denote service calls. When called, the results of these services are inserted in the document as siblings of the corresponding <sc> node. This process is called materialization.

Such documents can be large, and since the tree-structure of XML documents is verbose, there may be problems to manipulate them in main memory. Alternative ways of storing and managing these documents are needed. Additionally, one should be able to pose queries to stored 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.
This is because the activation of service calls may be associated to some query criteria, that is, a service may need to be called to answer a given XML query. Service calls may also be completely disassociated from queries. They may need to be activated periodically, 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. [3, 5] developed a platform to manage AXML documents. This platform is publicly available [9] and has strong “correction” properties, since it follows the AXML model, preserving document properties and types. Previous work on AXML materialization in this platform had mostly addressed typing control [25], XML query processing [1], and data and Web services replication [8]. In the first version of this platform, AXML documents were stored in the file system, which poses several drawbacks (security, indexing, etc.). Currently, AXML documents can be stored in the eXist [17] or Xyleme [42] XML DBMS. However, AXML materialization, i.e., query processing with service invocation is still processed apart from the DBMS. Thus, in this approach, queries need to be processed directly over those files, and documents still need to be loaded into main memory. This has serious scalability problems, especially when the documents are large. Thus, when storage and query capabilities are not within the same solution, the AXML document has to be manipulated by two different memory managers. An alternative approach would be to use a DBMS, since it provides both features, and, depending on how documents are stored, it can process queries without loading the documents entirely into main memory. This is an attractive solution for large documents, which are very frequent considering the verbose characteristic of XML. To use a DBMS two issues need to be addressed. One is the DBMS capability in storing and querying XML documents. The other is the DBMS ability in working with Web service calls.

There are several approaches to store and query XML documents in DBMS. Some use Relational DBMS [15, 19, 23, 32, 36, 38], others use native XML storage [21, 35]. However, the active part of AXML documents poses some challenges both to store and to 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). Relational
DBMS is able to support some dynamicity through SQL triggers. However, service calls may need to be activated at query time and triggers cannot 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 proposed solution, ARAXA\(^1\), uses Object-Relational (OR) DBMS. Although they cannot explicitly model the active behavior of AXML, OR-DBMS are capable of dealing with complex objects and associated methods. Methods can be seen as an active component. This allows us to create a class of active objects that are responsible for coordinating service calls and their execution. By using these resources, services can be called within SQL queries. It is also possible to create an agent that verifies the periodicity in which a service needs to be called, and manages these calls automatically. To support XQuery within OR-DBMS, we can use existing XML-relational storage mappings [16, 22, 24, 39], and consequently, existing algorithms that translate XQuery to SQL queries [22]. By using these algorithms together with our service call functions, queries can be processed with no need to load source documents into main memory, so very large documents can be processed efficiently without memory limitations. We focus on using standard resources in OR DBMS, so that our solution can be applied to any OR DBMS. In our solution, we keep the properties of the formal foundation of AXML documents [5, 6]. At the same time, we offer more sophisticated storage resources allied with consolidated query processing capabilities.

In summary, we have two main goals, where solutions and experimental results are our main contributions:

1) Scalability, which we address by managing query processing and service calls materialization in a single environment (the DBMS). This allows us to handle large XML documents without needing to load them into main memory. Processing an XML document in main memory is a requirement in previous solutions and has strong limitations even for small XML documents;

2) Single environment: We take advantage of DBMS algorithms to deal with materialization and query processing in a single environment (the DBMS itself), which contributes to improving the materialization process.

The limitation of our approach resides in the mapping between OR and XML. However, our results show a negligible overhead in this transformation. In fact, this is highly compensated by the fact that an organization can now keep their traditional data and AXML documents 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 paper is an extended version of a previous published paper [18]. In this paper, we detail our approach and present extensive experimental results. Moreover, we present a detailed description of our query processing methodology. This paper is organized as follows. Section 2 presents the Active XML

\(^1\) **ARAXA** is a Brazilian city and a Portuguese acronym that loosely translates to English means An object-Relational Approach to store XML Documents with Active elements.
Platform developed by the INRIA-GEMO group. Section 3 overviews related work and analyzes current solutions for storing and querying AXML documents. In Section 4 we identify the difficulties to the problem and propose 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 and its prototype. Our experimental results are discussed in Section 7. Finally, Section 8 concludes this work.

2. Background: Active XML

The Active XML Platform developed by the INRIA-GEMO group [9] is an open-source framework to support Active XML documents in a P2P distributed environment. Abiteboul et al.[2] defined a formal model for an AXML document where several materialization strategies can be applied [25, 28]. The materialization of Active XML data can be either explicitly requested by the user or implicitly triggered by queries that require the (materialized) content of a document.

![Active XML Architecture](image)

**Figure 2. Active XML Architecture [3]**

The internal architecture of an AXML peer, shown in Figure 2, relies on the following modules [9]:

- The *AXML storage*, 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 latter accordingly;
- The *XQuery processor*, which executes XQuery queries.

Peers communicate with each other only by the means 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 review some characteristics of the materialization of AXML documents as processed by the AXML platform. These specificities were defined by the AXML model [6]. Particularly, we discuss their approach in handling the active part of the documents. In Section 2.1 we show how
services are analyzed for query processing and then in Section 2.2 some materialization approaches are discussed. Notice that some of the approaches discussed here are currently not integrated to the Active XML platform.

2.1. 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 passed between a service call and its response may have significant differences from one service to another. Anyway, a service call 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 to minimize service calls would be to first 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 calls). Queries like /library/books/book[price > 90] over the document of Figure 1 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.

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

Figure 3. Example of LPQ
```
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 of the document in Figure 1. The set $S$ is generated by using each step of the path expression concatenated with *(). This represents the service calls.

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 $\text{library/books/book[price > 90]}$).

![Diagram](image)

**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 entitled “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.
2.2. Materialization Plans

Once we have the set of services that need to be called (see Section 2.1), 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 [34]: 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 (see Figure 5 (A) for a simplified example – the figure shows the service calls only). Materialization plans must respect these dependency types when defining the order in which services will be called.

The AXML document materialization processes presented in [28] use a service call dependency graph 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 [28]. 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 the followedBy attribute are represented by dotted arrows.

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

Figure 5. (A) AXML document; (B) dependency graph of (A) [28]

These algorithms have been proposed to be incorporated in the XCraft optimizer [33] for the Active XML platform. However they can be used independently of the platform.

3. Storing Active XML documents

In the literature, the main focus of work related to AXML documents has been the development of an initial infrastructure to support the execution of service calls and manage their results [4, 6, 7, 11, 14, 41]. In this way, the problem of storing large amounts of AXML documents has not received much attention.
The initial implementation of the Active XML Platform developed by the INRIA-GEMO group [9] stores AXML documents in a file system directory. This directory is application-defined, and no other storage alternative is provided. File system storage does not provide access control, indexing or data compression. In this way, we can anticipate problems with the management of stored documents, which can directly interfere in the scalability of the implementation and of the applications that use such documents. Another important limitation of this approach is on query processing. Since documents are stored in the file system, they need to be entirely loaded into main memory when queried.

To overcome such problems, the GEMO group proposed the Xyleme-AXML [6] and eXist-AXML [5], implementations of the AXML Peer integrated with the Xyleme Server [42] or eXist [17] (native XML repositories). In these approaches, the AXML document is stored as if they were regular (non-active) XML documents. The user application must deal with the management of the active part of the documents. An improvement of this approach is that XQuery queries are used to selectively access data. This way, documents are not entirely loaded into memory for query processing, but document materialization is still managed outside the DBMS. Additionally, native storage may not be the best alternative to enterprise applications, which usually store all of their data in RDBMS or OR-DBMS. 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 DBMS already in use in the company. The use of a native DBMS, in this case, would represent a considerable extra cost: maintenance of an integration model for DBMS with different paradigms; acquisition; and training. One of the main advantages of using relational (or object-relational) storage is its maturity and robustness. Results on XML storage has shown that object-relational DBMS are an efficient alternative to store XML [24].

In the previous paragraph, we claim that using an Object-Relational DBMS would contribute to improving the integration of legacy data with AXML documents. This is not, however, the approach taken by the AXML platform, which addresses this integration by using service calls. Of course services can be used to achieve integration, but having data in the same storage system makes things simpler. The key point is that all data is now kept in a single repository, sharing the same representation model and management tools Active and non active documents can be managed by the same DBMS. Thus, data administration procedures can be managed in an integrated way.

Mapping XML documents to relations is a well-covered topic in literature [16, 22, 24, 30, 39, 40]. 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. By using triggers, one can manipulate the dynamic properties of base data. They allow procedures to be automatically started based on the Event-Condition-Action (ECA) paradigm and/or 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 [12].

Nevertheless, to manage AXML documents stored in relations we need a class of triggers that is not implemented in most of the commercial DBMS. This class involves events on selections, that is, triggers...
started by SQL expressions like “select <columns> from <tables> where <predicates>”

Another specificity of AXML documents is that a service may be defined to be called in a timely manner (from time to time, or in a specific time). Existing 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, in the next section we present our proposal to store and manage AXML documents using object-relational systems and their complex types. Complex types allow us to associate methods to a given data type. Such procedures would be able to activate service calls.

4. Managing and Storing AXML Documents in ARAXA

ARAXA stores and manages AXML documents using an Object-Relational DBMS. AXML documents are stored in plain relations (no user defined types are used), but user defined types and methods are used to provide the dynamic features needed by such documents. We have created 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.

To store AXML documents, we studied existing approaches on XML-relational mapping and on XML-SQL query translation [15, 19, 20, 23, 32, 36, 38, 39]. Some of these approaches are schema dependant, that is, they generate relational schemas that are only able to store XML documents that conform to a given schema (which is used as input to the schema generation algorithm). The problem with such approaches is that, in our case, documents have a dynamic structure. The result of a service call may be heterogeneous, and using a schema-dependent mapping would imply on frequent schema modifications. Among the schema independent proposals in literature, we chose the mapping scheme proposed by Tatarinov [39, 40]. This approach defines a generic order-preserving schema to store XML documents that associates a numbering scheme to the nodes. The numbering scheme we use in our work is based on the Dewey encoding [26]. In this encoding, the root node of the document is assigned code “1”. Assume that the root node has two child nodes. They will be assigned codes “1.1” and “1.2” respectively, where the first part of the code points to the parent node code (which in this case is “1”). Codes of sibling nodes are consecutive numbers (1, 2, …). Children of node “1.1” are labeled “1.1.1”, “1.1.2”, …, and so forth (notice that they all have the same prefix “1.1”, that is the parent code). The left hand-side of Figure 6 shows an example of this numbering scheme. The main advantage of using Dewey encoding is that it minimizes the cost of reordering the document in cases of updates (insertions and deletions), since only siblings (and their sub-trees) of the updated node must be renumbered.
In the next section, we describe Tatarinov’s mapping scheme and the extensions we needed to make it more generic.

4.1. Mapping AXML documents to relations

Tatarinov et. al [39] propose to spread an XML document into two relations: Path (id, path) and Edge (dewey, path_id, value). In the Edge relation, the attribute dewey stores the dewey code of a node (as explained above). 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, and this can be used to speed up query processing. Notice that the Edge relation does not store the node name. It can be retrieved through the Path relation.

This mapping scheme consider elements, but not attributes [39]. Storing attributes as if they were elements would cause several problems. First, the document would be reconstructed in a wrong way, because attributes would become elements in the reconstructed document. Second, even if we mark attributes with an “@” in front of its name, a dewey number would be generated for each attribute. Such numbers would interfere in the reconstruction algorithm, since this approach does not guarantee that sibling elements would have consecutive numbers. Besides, this schema does not support the storage of several documents – the approach deals with only a single document, and thus the Edge relation has not a docId that would allow storing several documents.

![Figure 6. Extensions to the Tatarinov’s proposal](image)

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. Notice that paths in the Edge relation are also stored with the “@” symbol prior 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 propose here can be seen in Figure 6. Notice that the attribute named id is stored with Dewey code 1.1.1.@ in the Edge relation.

4.2. Storing Web service calls in object-relations

Once the mapping has been defined, our next goal is to find a way to manage the dynamic properties of the documents. For this, we need first to properly store information about service calls, which are responsible for the active part of a document. Notice that this information is indeed stored in the Edge and Path relations, but it is mixed with regular XML nodes, and thus it is difficult to manage service calls that way.

The first step is to identify service calls from regular nodes. This is relatively easy once elements names that represent service calls are standardized: <sc>. After identified, we store this information in a Service Call Catalog that is stored in the DBMS as two relations with the following structure:

\[
\begin{align*}
\text{Service\_call} & (id, \text{path\_id}, \text{dewey}, \text{doc\_id}, \text{serviceURL}, \text{methodName}, \text{serviceNameSpace}, \text{useWSDLDefinition}, \text{signature}, \text{callable}, \text{frequency}, \text{lastCalled}, \text{followedBy}, \text{mode}, \text{doNesting}) \\
\text{Parameter} & (id, \text{service\_id}, \text{path\_id}, \text{type}, \text{name})
\end{align*}
\]

The Service_call relation stores all service calls within a given AXML document. It also stores two additional information: the document id in which they appear (doc_id); and where they are located within the document (path_id). This relation also stores the service call attributes (serviceURL, serviceNameSpace, methodName, signature, useWSDLDefinition, id, name, callable, frequency, lastCalled, followedBy, mode, doNesting), 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

\[
\text{select execute\_service (doc\_id, service\_id, dewey)}.
\]

In this select clause, execute\_service() is a function added by our mapping strategy that calls a generic client method for Web Services that is able to activate service calls.
This generic Web Service client was implemented in Java outside the DBMS (the right hand-side of Figure 7 shows a simplified class diagram of our implementation), so no modification to the DBMS core was made. Instead, we associated the implementation of some methods of this client with the OR-DBMS by using user-defined functions (UDF). This association process is available in most of the OR-DBMS, since they support high-level programming languages.

```
-- example of function/method association in ORACLE --
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;
```

```
-- example of function/method association in DB2 --
CREATE PROCEDURE execute_service
(IN serviceld INTEGER,
IN dewey VARCHAR,
IN docld INTEGER)
EXTERNAL NAME 'axml.Active.executeService'
RESULT SETS 0
LANGUAGE JAVA
PARAMETER STYLE JAVA
FENCED NO
DBINFO NULL
CALL MODIFIES SQL DATA;
```

**Figure 7. Class diagram of the AXML storage components**

In Figure 7 we show the associations between methods of our implementation and the UDFs we defined in the database schema. Two UDFs are added to the database schema by our mapping strategy: execute_service() and start(). The execute_service() function is shown at the upper-left box of Figure 7. Basically, it calls the executeService() method of the Active class of our external Java implementation. This method is responsible for calling services and delegating important tasks related to materialization and mapping of results to other components of our approach. The start() function (lower-left box in Figure 7), is associated with the start() method of the MonitorAgent class. This method 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. In Figure 7, the UDFs use the syntax of PostgreSQL, however, similar mechanism are available at Oracle 9i [27], IBM DB2 [37], among others. Figure 8 shows some examples.

We want to emphasize 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 6.1. 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 OR-DBMS.

5. A methodology to process queries in ARAXA

Executing a query on AXML documents may involve materializing active elements. There are several alternatives to execute the service calls needed by the materialization process. Optimization strategies have been proposed for such materialization while preserving the document properties [1, 28]. 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 only the services that need to be executed. Ruberg et al. [28] show how to extract the dependencies on these service executions and present a dependency graph generator. Based on this graph they propose optimization strategies for these service executions.

To process queries over AXML documents in ARAXA, we take advantage of those previous successful techniques by adapting them to our storage structure. Based on them, we have defined a methodology to process queries with service calls on AXML documents using ARAXA. Given an XQuery \( q \) over stored AXML documents, we process it using the following steps of our methodology:

(i) to identify services that need to be called to answer a query \( q \);
(ii) to translate XQuery query \( q \) to an SQL query \( q' \);
(iii) to identify the dependencies among service calls in \( q \);
(iv) to define the calling order and call the services using our UDFs execute_service() and start() (see section 4.3);
(v) to store service call results in the Edge and Path relations;
(vi) to execute query \( q' \); and
(vii) to map the resulting tuples to XML and return the answer to the user.

To explain the steps of the methodology we will use the example on Figure 9. The example document (Figure 9 (A)) 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). The example query (Figure 9 (B)) retrieves the price of the book with ISBN = “12345”. We now show how the steps of our methodology are used to answer this query \( q \).

In step (i), we analyze what services are relevant to answer \( q \). 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 proceed this identification (see details on Section 2). In the example of Figure 9, we find out that the service on node “1.1.2.1” needs to be called.

Step (ii) translates the XQuery/XPath query to SQL using the algorithm sketched by Tatarinov et al. [39]. In our running example, the translation is shown in Figure 9 (B).

During step (iii) we use the dependency graph generator from [28] to generate a materialization plan for the query that respects the dependencies between service calls (the dependencies are identified at document storage time – see Section 6 for details). Based on this plan, we generate the SQL queries that will actually call the services by using the execute_service() function. The parameters needed for the service call are taken from the catalog. Notice that, as a result of this step, we may have a set of SQL queries. In [28], the materialization plan includes delegation of service execution’s control to nodes in a P2P network, with a Master Site orchestrating only the initial execution. In our approach, however, we do not use delegation. The DBMS plays the role of the Master Site, and it is entirely responsible for the service execution orchestration. Thus, the SQL statements generated in step (iii) do not take delegation into account. In our example, since we have only a single service to be called, a single SQL statement will be generated: select execute_service(12, “1.1.2.1”, 1) (see the top of Figure 9 (C)), where 12 is the service_id in the catalog, “1.1.2.1” is the dewey code of the service node in the document, and 1 is the doc_id.

Figure 9. Example the query translation

Step (iii) generates several SQL queries, one for each service that needs to be called. In the step (iv) of our approach, we use Ruberg’s SLS algorithm [28] to define an optimized execution order to the service calls. Basically, this step orders the SQL statements generated in the previous step. Service calls that have any execution order restriction need to be executed by distinct SQL queries that will be sent to the DBMS in the correct order, as shown bellow.
select execute_service(...);
select execute_service(...);

On the other hand, service calls that can be executed in parallel are defined in a same query. In the example below, we call two services (in the same SQL query). Such services can be executed in parallel, so we do not impose any restriction in the order in which they are called.

select execute_service(...), execute_service(...);

As a result of this step, we produce a SQL script that contains several SQL queries. Queries that need to be executed in a certain order are put in separate queries, respecting the order in which they must be executed. These SQL statements are then sent to the DBMS and executed. In our running example, a single SQL statement will be executed in this step.

In step (v), the results of each service execution are inserted in the stored AXML document using the XML-rational mapping rules. The results of each service call are embedded in a <result> element, which is inserted as the immediate right sibling of the corresponding <sc> element in the stored document. The executeService() method called by the execute_service() UDF function calls methods that are responsible for mapping the <result> subtree into tuples and inserting them in the Edge and Path relations.

This is why services are called (in step (iv)) before the SQL query that corresponds to the user XQuery is executed. In fact, execute_service() statements are not sub-queries of the translated XQuery query q’. This is not necessary, since execute_service() statements modify the database state (the result of the service calls are stored in the Edge and Path relations). In this step, the Service Call Catalog is also updated (it contains information such as time of last execution of a given service, among others).

Step (vi) executes the translated query q’. In this step our approach benefits from the DBMS query engine. In our example, query q’ is the second query in Figure 9 (C).

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. In our example, the query result is <price>23.4</price>. Notice that there is a post-processing step applied to the query result. The result returned by the XPath query is actually the subtree rooted at node “1.1.2” in Figure 9 (A). The post-processing step removes the <sc> subtree and merges the result node content into its parent (price).

6. Architecture and Prototype of ARAXA

The storage scheme, service call activations and query processing methodology shown in the preceding sections are implemented in ARAXA according to the architecture shown in Figure 10. The architecture is composed of two main modules: the Control Module and the Integration Module. They are further divided into sub-modules.
The Integration Module (left hand-side of Figure 10) is responsible for mapping AXML documents to relations and for processing queries. This module 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 on the Service Call Catalog of the Control Module. All the information needed to generate the service calls dependency graph is also obtained at this stage and stored in the Service Call Catalog. This will be needed at query execution time.

The Query Translator/Processor module translates an XQuery/XPath to SQL and identifies the services that need to be called to answer the query. It is also responsible for sending the SQL query to be executed in the DBMS, and mapping the resulting tuples back to XML. Our current implementation supports XPath only. We plan to add XQuery support in near future.

![Figure 10. Architecture of ARAXA](image)

The Integration Module is ARAXA’s interface with the user. This means that all the remaining details are hidden. The user is not aware of how documents are stored or how services are activated. This transparency is provided by the Control Module. It is composed of the Service Call Catalog, Service Manager, Result Manager and Monitor Agent.

The Service Call Catalog stores information about the service calls embedded in the document (<sc> subtrees). This information includes: the behavior defined by the document designer; service call criteria; activation parameters; service call location within the document; statistics about service execution and service providers; and information needed to construct the service calls dependency graph. The Catalog is populated with information extracted during the XML-Relational mapping, and then fed by other architecture components during the system’s activity. The Catalog provides information to other ARAXA’s 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()` UDF. This component first verifies whether the service really needs to be called. If so, it gets the service’s parameters from the Service Call Catalog and calls the service by communicating with the external
environment. Once the external service provider returns the result, the Service Manager passes it to the Result Manager.

The Result Manager is responsible for materializing the result of a service call within the mapped AXML document. To do so, it applies to the resulting XML tree the same XML-Relational mapping used to store the original document. The AXML model defines two distinct materialization behaviors: replace or append. This is defined at document design time by an attribute called mode. In our approach, this information is stored in the Service Call Catalog. The use of the replace behavior means that the old service call result will be replaced by the new one within the AXML document. The append behavior appends the new result next to the previous one (this is the default behavior). After materializing the result by using one of these behaviors, the Result Manager updates the Service Call Catalog (last time the service was called, time of response, etc.).

All the components we described previously are responsible for some steps of our query processing methodology. However, we still need a very important component that will be responsible for activating service calls independently from queries. The Monitor 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 date/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.

As shown in this section, the Integration and Control modules are both responsible for our query processing methodology presented in the previous section. More specifically, the Integration Module is responsible for steps (i), (ii), (iii), (iv), (vi) and (vii). The remaining step (v) is executed by the Control Module.

6.1. Prototype

We have developed a prototype of our approach using PostgreSQL [31] as our Object-Relational DBMS. The Control Module was implemented internally to the DBMS using PLJava [29] 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 translator was developed by Medeiros and Taok as their undergrad final project.
We have implemented two strategies for selective query evaluation: filters and LPQ. We have not implemented optimizations to the materialization plans yet. However, we do respect the dependencies between service calls. In the next versions we intend to integrate ARAXA with the XCraft optimizer [33].

We have also implemented a user interface. Through this interface, a user can select AXML documents to be stored and pose queries over them. We have used this interface to run our experiments.

We are also working on integrating our prototype with the Active XML platform. The Gemo group developed an API that could be used to provide this integration.

7. Experimental Evaluation

In this section we present the results obtained from experiments with the ARAXA prototype. The main goal of our experimental evaluation is to analyze the viability and scalability of our approach, together with an analysis of the coupling of active data management strategies in OR-DBMS. We also evaluate the impact of mapping XML to OR in the context of querying and materialization of AXML documents. This is done in Section 7.2. However, in order to show that memory-based solutions (like the AXML platform) have difficulties in processing large XML files, we start in Section 7.1 by analyzing the size occupied by XML documents when represented as DOM trees in main memory. Finally, Section 7.3 shows an evaluation of query processing in OR-DBMS versus Native XML DBMS.

7.1. Large XML files in Main Memory

We have measured the size occupied by different XML files in main memory, when parsed as DOM trees. To measure the memory size, we have used the Java Runtime.getRuntime().getTotalMemory() and Runtime().getRuntime().freeMemory() methods. The getTotalMemory() method retrieves the total amount of memory allocated by the Java application, while the freeMemory() method retrieves the amount of allocated memory that is free. The difference between total memory and free memory gives us the amount of memory that is actually being used by the Java application. Thus, to measure the size of the DOM tree, we collected the used memory before and after calling the parse method (which builds the DOM tree in memory), and then subtracted the two values. Table 1 shows the obtained results. Files named subversion and apache where generated with the –xml option of svn log. They represent logs of the commits to the subversion and apache projects respectively and where generated with different options (no option, -v and -q). File XMark was obtained from the XMark benchmark [13] and DBLP from the DBLP XML website. Docs1, 2, 3, 4, 5 and 6 are the docs we used in our experiments of query processing. They are described in the next section.

The numbers in Table 1 shows an increase of at least two times in the size of the file when loaded in memory using DOM. The tests were executed in a Intel Core2 DUO P9500, 2.53GHz, with 3GB RAM. To be able to process files larger than 20MB, we needed to increase the amount of memory available to the application (with the –Xmx parameter). Even using Java settings to increase the amount of available memory to the application (with the -Xmx parameter), the two larger files (of 535.3MB and 904.7MB)}
could not be processed. They both raised an “out of memory” error. Notice that the DBLP file could not be processed by SAX either for the same reason (we used SAX to count element, attributes and text nodes of the documents). This fact, which seems unusual, is caused by the excessive number of entity expansions required to parse the file (we had to increase the entity expansion limit using the \( \text{–Dentityexpansionlimit=560000} \) parameter in the Java application).

Table 1. XML File size X DOM tree size

<table>
<thead>
<tr>
<th>File Name</th>
<th># of Elements</th>
<th># of Attributes</th>
<th># of text Nodes</th>
<th>File Size (MB)</th>
<th>DOM Size (MB)</th>
<th>Tree Size increase</th>
</tr>
</thead>
<tbody>
<tr>
<td>Subversion1.xml</td>
<td>97887</td>
<td>32800</td>
<td>195773</td>
<td>3.60</td>
<td>21.22</td>
<td>589.5%</td>
</tr>
<tr>
<td>Subversion2.xml</td>
<td>130656</td>
<td>32800</td>
<td>373011</td>
<td>14.60</td>
<td>54.48</td>
<td>373.19%</td>
</tr>
<tr>
<td>Subversion3.xml</td>
<td>286011</td>
<td>165571</td>
<td>683736</td>
<td>25.10</td>
<td>101.66</td>
<td>405.04%</td>
</tr>
<tr>
<td>Apache1.xml</td>
<td>2054701</td>
<td>690777</td>
<td>4109215</td>
<td>76.60</td>
<td>443.40</td>
<td>578.86%</td>
</tr>
<tr>
<td>Apache2.xml</td>
<td>3051116</td>
<td>769792</td>
<td>6611903</td>
<td>160.3</td>
<td>791.66</td>
<td>493.86%</td>
</tr>
<tr>
<td>Apache3.xml</td>
<td>9569246</td>
<td>12922748</td>
<td>19648655</td>
<td>904.7</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>XMark.xml</td>
<td>1666315</td>
<td>381878</td>
<td>3026904</td>
<td>113.7</td>
<td>444.33</td>
<td>390.79%</td>
</tr>
<tr>
<td>Dblp.xml</td>
<td></td>
<td></td>
<td></td>
<td>535.3</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>Doc1</td>
<td>165</td>
<td>70</td>
<td>331</td>
<td>0.09</td>
<td>0.43</td>
<td>461.74%</td>
</tr>
<tr>
<td>Doc2</td>
<td>129</td>
<td>62</td>
<td>833</td>
<td>0.09</td>
<td>0.39</td>
<td>421.32%</td>
</tr>
<tr>
<td>Doc3</td>
<td>1707</td>
<td>490</td>
<td>1959</td>
<td>9.9</td>
<td>19.80</td>
<td>200.05%</td>
</tr>
<tr>
<td>Doc4</td>
<td>64772</td>
<td>370</td>
<td>59520</td>
<td>9.7</td>
<td>22.97</td>
<td>236.82%</td>
</tr>
<tr>
<td>Doc5</td>
<td>5104</td>
<td>980</td>
<td>10301</td>
<td>96.4</td>
<td>189.49</td>
<td>96.56%</td>
</tr>
<tr>
<td>Doc6</td>
<td>68365</td>
<td>70</td>
<td>67341</td>
<td>97.7</td>
<td>195.67</td>
<td>200.28%</td>
</tr>
</tbody>
</table>

The main strength of our approach is to be able to handle large XML files, since there is no need to load the documents in memory to process queries. Since documents are stored in the DBMS, queries are directly translated to SQL and no main-memory XML manipulation is needed. More precisely, in ARAXA, the amount of memory needed to process a query does not depend on the size of the document. In the next section, we evaluate query processing and document materialization in ARAXA.

7.2. Query Processing and Document Materialization in ARAXA

Our main experiments were focused on query processing and document materialization. We evaluated the time spent in each step of our query processing methodology and also the impact of our approach in the behavior and performance of the OR-DBMS. We did not compare our execution time with the Active XML platform, once their storage system is different, so we would have no comparison basis.

We have also not measured the mapping time needed to store the document in relations. We took this decision because this mapping occurs only once, and thus it is not a critical factor of our approach.

The tests were executed in an Intel(R) Core(TM)2 CPU T5300 1.73 GHz with 2038MB RAM memory running Windows Vista Home Premium Edition and PostgreSQL 8.2, with no special tuning or configuration. We also used the JRE1.6.0_02 Java virtual machine, using buffer size of 8192 bytes and java heap size of 128Mb (both are the default values of JDK). For the Web Service infrastructure, we have used jakart-tomcat-4.1.31. Services were provided through SOAP RPC.

To generate the AXML documents, we have used ToXgene [10]. We modified the XMark and Catalog templates of ToXgene, and customize them to include service calls in the documents to be
generated. We also made some modifications on the templates to generate the size and structure variations we needed.

In our experiments, we explore six distinct documents. Over each of these documents we execute XPath queries and evaluate the performance of each step of the query 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 defined by a service call. The service call uses the book ISBN as the input parameter. We now describe each of the six documents:

- **Doc1** (highly structured): a catalog with 5 books and 5 magazines; 10 service calls to retrieve the price of book or magazine; size of 97KB.
- **Doc2** (loosely structured): an item list with 1 book and 9 heterogeneous items; 1 service call to retrieve a bonus book chapter (that changes over time); 9 service calls to retrieve delivery information; 1 service call to retrieve the book price; size of 96KB.
- **Doc3** (highly structured): a catalog with 35 books and 35 magazines; 70 service calls to retrieve book or magazine prices; size of 10MB.
- **Doc4** (loosely structured): an item list, where 10 of them are books; 60 service calls to retrieve delivery information; 10 service calls to retrieve book prices; size of 10MB.
- **Doc5** (highly structured): a catalog with 70 books and 70 magazines; 140 service calls to retrieve book or magazine prices; size of 96MB. Notice that this document is large and difficult to manipulate in main memory.
- **Doc6** (loosely structured): an item list, where 10 of them are books; 130 service calls to retrieve delivery information; 10 service calls to retrieve book prices; size of 98MB.

These documents were chosen to evaluate several aspects: Documents 1 and 2 are small and have few service calls. They can easily be handled in main memory. Documents 3 and 4 are larger and have 7 times more service calls, which we believe is hard to manage in main memory only. Documents 5 and 6 aim at representing large documents that are hard to manipulate in main memory. The set of documents we used in our experiments reflect real XML documents, since it contains both highly structured and non-structured documents. Also, ToxGene has been largely used to generate test data, so the generated documents are reproducible.

<table>
<thead>
<tr>
<th>Query</th>
<th>Document</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1: <code>/catalog/book/price</code></td>
<td>Doc1, Doc3 and Doc5</td>
</tr>
<tr>
<td>Q2: <code>/item_list/book/price</code></td>
<td>Doc2, Doc4 and Doc6</td>
</tr>
<tr>
<td>Q3: <code>/catalog/book[@isbn=&quot;5950193442&quot;]</code></td>
<td>Doc1, Doc3 and Doc5</td>
</tr>
<tr>
<td>Q4: <code>/item_list/book[@isbn=&quot;7813071809&quot;]</code></td>
<td>Doc2, Doc4 and Doc6</td>
</tr>
<tr>
<td>Q5: <code>/item_list/book[@isbn=&quot;7813071809&quot;]/freeChapter</code></td>
<td>Doc2</td>
</tr>
</tbody>
</table>

Table 2. Queries and the documents they were applied to
The queries we used to evaluate our system explore the LPQ and filter approaches. They are shown in Table 2 together with the documents over which we applied them. To evaluate our approach with LPQ, we executed queries Q1 and Q2. Queries Q3, Q4 and Q5 were used to evaluate our approach with the filters strategy. Each query was executed 10 times on the documents shown in Table 2. The results we present here are the means of these executions. In the graphics, we show the complete execution time and discriminate the time spent on specific tasks according to our methodology (Section 5) as follows:

1. XPath to SQL translation
2. 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 LPQ)
3. 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 generating the SOAP message
4. 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.
5. 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).
6. Execution of SQL query – step (vi) of our methodology.
7. Tuples to XML translation – step (vii) of our methodology (translate the relational query answer back to XML).

Among these steps, we can notice three that represent the overhead of our approach: steps 1, 5 and 7. Steps 1 and 7 present mappings and translations needed due to the change of paradigm (from XML to relations). Step 5 corresponds to the materialization of service call results. This is, however, a required step of any AXML query processing, independently of how the documents are stored. In our case, however, we need to apply the mapping algorithm to store the service call results in the Edge and Path relations. Because of this, we assume this step as an overhead of our approach, even though only part of it is truly an overhead. We use these three steps to measure the overhead of our approach, since the remaining ones all have corresponding steps in any other approach.

In the next sections we show our experimental results.

**7.2.1. Queries without filters (Q1 and Q2)**

Figure 11 shows the results of query Q1 (which has no filter) applied over the highly structured documents. To process this query, the LPQ strategy was used.

We can notice that most of the query execution time was spent waiting for the Web Service execution response. It is important to notice that this would happen in any other storage approach. This time is large
because of external factors such as service provides availability, network environment and message exchange. The LPQ strategy was able to eliminate all irrelevant service calls (the ones that retrieve magazine prices - 5 in Doc1, 35 in Doc3 and 70 in Doc5).

The overhead of ARAXA is low in all of the cases. Doc5 has the highest overhead (18.3%) due to the size of the query result. For this document, step 7 takes 16880ms to execute. This is expected since the query retrieves half of the 96MB document, so a large portion of XML data has to be generated in this step. Notice that executing this kind of query in a memory-based approach would also take a long time.

Table 3 summarizes the results for Q1. It shows the actual number of services called and the execution time of the services. Notice that the service execution time is responsible for almost 80% of the total query processing time. More importantly, the table shows the total ARAXA overhead time in each case. Based on these values, Figure 12 shows the increase in the ARAXA overhead time when the size of the document increases. We used the smallest document (Doc1) as basis for the measures. The figure shows the scalability of our approach for Q1. To improve the readability of the figure, we use logarithmic scale. Notice that the real differences are much more evident than that shown in the figure.

Table 3. Summary data for Q1

<table>
<thead>
<tr>
<th>Doc</th>
<th>Size (KB)</th>
<th>Total Query Execution Time (ms)</th>
<th>Service Execution Time (ms)</th>
<th>% (service execution time over total execution time)</th>
<th>Total ARAXA overhead time (ms)</th>
<th>Number of Services Called</th>
</tr>
</thead>
<tbody>
<tr>
<td>Doc1</td>
<td>97</td>
<td>7801</td>
<td>5820</td>
<td>74,60581977</td>
<td>795</td>
<td>5</td>
</tr>
<tr>
<td>Doc3</td>
<td>9960</td>
<td>49510</td>
<td>40775</td>
<td>82,35709958</td>
<td>4086</td>
<td>35</td>
</tr>
<tr>
<td>Doc5</td>
<td>96499</td>
<td>102977</td>
<td>81483</td>
<td>79,12737796</td>
<td>19580</td>
<td>70</td>
</tr>
</tbody>
</table>
Figure 12. Scalability of ARAXA for Q1

Figure 13. Results of Q2

Table 4. Summary data for Q2

<table>
<thead>
<tr>
<th>Doc</th>
<th>Size (KB)</th>
<th>Total Time (ms)</th>
<th>Service Execution Time (ms)</th>
<th>% (service execution time over total execution time)</th>
<th>Total ARAXA overhead time (ms)</th>
<th># of Services Called</th>
</tr>
</thead>
<tbody>
<tr>
<td>Doc2</td>
<td>96</td>
<td>1801</td>
<td>1164</td>
<td>64.63076069</td>
<td>277</td>
<td>1</td>
</tr>
<tr>
<td>Doc4</td>
<td>9743</td>
<td>18593</td>
<td>11670</td>
<td>62.76555693</td>
<td>899</td>
<td>10</td>
</tr>
<tr>
<td>Doc6</td>
<td>97722</td>
<td>23642</td>
<td>11690</td>
<td>49.44590136</td>
<td>2365</td>
<td>10</td>
</tr>
</tbody>
</table>
proportionally smaller than that of Q1. Still, our overhead is low. The scalability factor is shown in Figure 14 (logarithmic scale).

![Figure 14. Scalability of ARAXA for Q2](image)

7.2.2. Queries with filters (Q3, Q4 and Q5)

Queries Q3, Q4 and Q5 were used to evaluate our approach when the filters strategy is applied. They all retrieve a subtree with information of a specific book. In this case, the filters strategy was able to eliminate all irrelevant service calls.

![Figure 15. Results of Q3](image)

Figure 15 shows the results of Q3 over the highly structured documents. Again, step 4 takes more time to execute than the other steps. In Doc3, the overhead of our methodology represents 23.4% of the total query execution time, which is relatively high. However, the total query execution time is approximately 3 seconds, which is low when we consider queries that involve web service executions. Thus, the total query time remains acceptable, even with our overhead.
In Doc5, there was a considerable increase in the time spent in step 1, which represents XPath to SQL translation. This is because during this step we execute some queries over the document (to define the document’s depth and ordering criteria). These queries impact increases when the amount of data to be queried is large (as is the case with Doc5). This means that we need to try to optimize the translation algorithm.

Table 5. Summary data for Q3

<table>
<thead>
<tr>
<th>Doc</th>
<th>Size (KB)</th>
<th>Total Time (ms)</th>
<th>Service Execution Time (ms)</th>
<th>% (service execution time over total execution time)</th>
<th>Total ARAXA overhead time (ms)</th>
<th># of Services Called</th>
</tr>
</thead>
<tbody>
<tr>
<td>Doc1</td>
<td>97</td>
<td>1649</td>
<td>1164</td>
<td>70.58823529</td>
<td>276</td>
<td>1</td>
</tr>
<tr>
<td>Doc3</td>
<td>9960</td>
<td>2007</td>
<td>1163</td>
<td>57.94718485</td>
<td>470</td>
<td>1</td>
</tr>
<tr>
<td>Doc5</td>
<td>96499</td>
<td>2998</td>
<td>1165</td>
<td>38.85923949</td>
<td>1139</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 5 presents the summary data for Q3. In all cases, only a single service was called. Even when the size of the document increases, total query time remains relatively low. Figure 16 shows the scalability of ARAXA for Q3 (logarithmic scale).

![Figure 16. Scalability of ARAXA for Q3](image)

In Figure 17 we show the results of Q4 applied over the loosely structured documents. For Doc2, the execution times obtained by Q4 are proportional to those obtained by Q2, since in both cases, only a single service was called. Notice that the overheads in this case are also similar (15.3% in Q2 and 15% in Q4).

Doc6 had the highest overhead of our experimental evaluation. This was due to step 1, for the same reasons we explained before. However, it is important to notice that even with large variations in the document sizes, there was a very small variation in the total query execution time (less than two seconds, as shown in Table 6).

Table 6 presents the summary data for Q4. Notice the relation between document size increase and total query processing time. Figure 18 shows the relation of document size increase and the ARAXA overhead (logarithmic scale).
Table 6. Summary data for Q4

<table>
<thead>
<tr>
<th>Doc</th>
<th>Size (KB)</th>
<th>Total Time (ms)</th>
<th>Service Execution Time (ms)</th>
<th>% (service execution time over total execution time)</th>
<th>Total ARAXA overhead time (ms)</th>
<th># of Services Called</th>
</tr>
</thead>
<tbody>
<tr>
<td>Doc2</td>
<td>96</td>
<td>1781</td>
<td>1164</td>
<td>65,35654127</td>
<td>267</td>
<td>1</td>
</tr>
<tr>
<td>Doc4</td>
<td>9743</td>
<td>2435</td>
<td>1167</td>
<td>47,92607803</td>
<td>276</td>
<td>1</td>
</tr>
<tr>
<td>Doc6</td>
<td>97722</td>
<td>4345</td>
<td>1164</td>
<td>26,78941312</td>
<td>1214</td>
<td>1</td>
</tr>
</tbody>
</table>

In Figure 19 we show the execution of Q5 over Doc2. Query Q5 retrieves the free-chapter of a given book. In this query, the overhead of our approach is 4.5%. This is the lowest overhead we had in our experiments. This is because the size of the free-chapter service call response is significantly larger than the price one. This implies in a larger data transferring time.
This query shows that the services and network features interfere in the overhead of ARAXA. Once the time of step 4 is large, the overhead tends to be small. Notice that this would be the case in most of the real scenarios. Our tests were carried out in the worst possible scenario to our approach: dedicated service providers. In most of the real cases, this won’t happen. This also explains why the service provider answers calls in similar times (see for instance Table 6) -- it is completely dedicated to answering ARAXA’s requests, which means there are no other (external) service calls to answer.

Another important point is that the DBMS connects to the service provider through an Internet link, but this does not represent any restriction on bandwidth. This is because the transferred data volume is small in most of the cases, and there is no bottleneck in the link between the DBMS and the service provider. This can be noticed in the behavior of Q5, which produces a larger data volume when compared to Q3. Q5 and Q3 are similar queries, but the result size impacts ARAXA’s overhead significantly (15.3% in Q3 x 4.5% in Q5). We conclude that the reason for this large difference in the overhead is the service call result size.

Another criterion we analyzed in our experiment was the Monitor Agent. We have made tests both with the Monitor on and off. In Figure 20, 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 20, we can observe that, even after the Agent startup, there was no change in memory usage on the DBMS.
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 order in which the services need to be called and use low-cost equivalent services [28]. 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 two of them (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.

### 7.3. Query Processing in OR-DBMS x Native XML DBMS

After evaluating ARAXA and its overhead, we decided to compare query processing in OR-DBMS (as ARAXA does) and in a Native XML DBMS. In this evaluation, we have kept ARAXA’s query processing methodology. We used ARAXA algorithms to parse XPath queries, convert them to SQL, execute them in the DBMS, and then convert the resulting tuples back to XML. We then compare the total execution time with the time the Native DBMS takes to process the query. Notice that, in this evaluation, we are not dealing with service calls. Thus, some of the queries may bring AXML service tags in the result. This is because so far native XML DBMS cannot handle service calls during query processing, as we mentioned before.
Figure 21. Results of OR-DBMS x Native XML BDMS
The tests were executed in an Intel(R) Core(TM)2 CPU T5300 1.73 GHz with 2048MB RAM memory running Windows Vista Home Premium Edition and PostgreSQL 8.4, with no special tuning or configuration. As the Native DBMS, we used eXist [17].

The test data of this experiment is the same we used in Section 7.2 (doc1, doc2, doc3, doc4, doc5 and doc6). These docs were submitted to a subset of the queries we used in Section 7.2 (Q1, Q2 and Q3 presented on Table 2). This subset of queries is enough to make our point. Queries Q1, Q2 and Q3 were applied to the same documents we used in the previous evaluation. Table 2 shows the details.

Each query was run 11 times. We have discarded the first run and taken the mean of the 10 remaining execution times. Figure 21(a) (b) and (c) shows the average execution time for each query, in milliseconds.

As expected, our results show that query processing in native DBMS is much more efficient than in OR-DBMS. ARAXA could greatly benefit from this strategy. However, as we argue in this paper, native DBMS does not provide mechanisms that could be used to call services, and thus are not suitable for our approach. If, in the future, they offer such mechanism, then using them in our approach will be a good option in terms of performance. Of course other requirements would need to be taken into account when moving towards native DBMS. For instance, if having a unique repository for relational and XML data is important for the company, this change won’t be a good choice.

8. Final Remarks

In this paper, we present ARAXA, a solution to store and manage Active XML (AXML) documents. We analyzed the features of this new data management, and built a solution that uses OR-DBMS. OR-DBMS, as well as R-DBMS and Native XML DBMS, do not provide explicit mechanisms to manage web service calls. However, we could find an alternative solution by linking UDFs and SQL procedures with user-defined object methods. Thus, we propose to store AXML documents in relations, and take advantage of query processing and complex object representation of the Object-Relational paradigm, which is robust and used in most commercial applications. To map AXML to relations, we use and extend the efficient approach from [39]. 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 implements this architecture.

Our experimental results show that our approach is scalable, but 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 and query 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. This way, 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 autonomic computational environments.

Acknowledgements. This work was partially funded by CNPq and INRIA. Vanessa Braganholo was partially sponsored by FAPERJ. The authors are thankful to the INRIA Gemo group for the ActiveXML prototype and are deeply grateful for Gabriela Ruberg’s discussions with respect to ARAXA and many tips on using and understanding the code of the Active XML platform. The authors also thank Medeiros and Taok for the implementation of the query translator, and to Leonardo Murta for helping with Java execution metrics and parameters.

References


18. Ferraz, C., Braganholo, V. and Mattoso, M., Storing AXML documents with ARAXA. in *Brazilian Symposium on Databases (SBBD)*, (João Pessoa, PB, Brazil, 2007), 255-269.


23. Lee, D. and Chu, W., Constraints-Preserving Transformation from XML Document Type Definition to Relational Schema. in *International Conference on Entity Relationship (ER)*, (Salt Lake City, Utah, USA, 2000), 323-338.


28. Pereira, D., Ruberg, G. and Mattoso, M., Generating efficient materialization plans for Active XML Documents (Geração Eficiente de Planos de Materialização para Documentos XML Ativos). in Brazilian Symposium on Databases (SBD), (Brazil - In Portuguese, 2006), 236-250.


36. Shanmugasundaram, J., Tufte, K., Zhang, C., He, G., DeWitt, D.J. and Naughton, J.F., Relational Databases for Querying XML Documents: Limitations and Opportunities. in International Conference on Very Large Databases (VLDB), (Edinburgh, Scotland, UK, 1999), 302-304.


38. Suciu, A.D.M.F.D., Storing Semistructured Data with STORED. in International Conference on Management of Data (SIGMOD), (Philadelphia, Pennsylvania, USA, 1999), 431--442.


