ABSTRACT

XML files are ideal for describing semi-structured data. The advent of native XML databases has enhanced storage of a large number of XML files preserving the hierarchical format. The challenge is to serve and access XML data from geographically distributed native XML databases. In this paper, we discuss the modus operandi for insertion and querying of data in a location transparent manner in decentralised networks. We present the design for a cluster based architecture where query processing is achieved using distributed hash tables. A bulk insertion method has been employed for insertion of XML data. The proposed design was implemented on eXist-db, a native XML database engine. Experiments were conducted and the effectiveness of the proposed design was evaluated.

Categories and Subject Descriptors
C.2.4 [Distributed Systems]: Distributed Databases; H.2.4 [Information Systems]: Database Management Distributed Systems

General Terms
Design, Experimentation

Keywords
Distributed XML Databases, Native XML Databases, Bulk Insertion, Distributed Query Processing

1. INTRODUCTION

The sheer volume of XML data and its exchange over the Internet has increased the need for efficiently storing and managing XML documents in a database. XML databases can be categorised as – native XML databases and XML enabled databases. Typically, native XML databases are preferred over traditional databases as they reduce data conversion efforts. Businesses and organisations have branches that are spread across various geographical locations, resulting in decentralisation of data. Each branch maintains its own local database with limited access to it. In order to obtain results from a distributed system, we need to query and collate data from multiple sources in a location transparent manner, i.e., the exact location of the XML files is abstracted from the user. Such decentralisation has generated considerable interest in processing queries in distributed XML databases.

There are significant bodies of work which have attempted to distribute XML databases. Research has been undertaken by Toshiba to develop a distributed native XML DBMS in order to provide real time access to large volumes of XML data. The state of the art is to fragment XML data based on certain schemes viz. i) Horizontal fragmentation ii) Vertical fragmentation, and iii) Hybrid fragmentation as discussed in [8, 16, 10, 13]. Subsequent schemes for a top-down approach for data fragmentation and transparent query processing are also suggested. Moreover, the approaches used in processing queries on dynamic XML documents on the Web can be incorporated in distributed XML databases as well. In [8], Guilherme et al, decompose queries into sub-queries which work on fragments of XML data but the system is specific to the inherent indexing mechanism.

This paper aims at building a more generic solution by having a distribution layer over the extant architecture of a native XML database. We present a cluster based protocol for a distributed database system, which has not been implemented before on native XML databases as per our knowledge. We propose a distributed query processing technique for this architecture which efficiently retrieves data from distributed data sources using various support data structures, primarily Distributed Hash Tables (DHT). We also present a specialised bulk insertion method which dynamically inserts data in a periodic fashion. Finally, we show how our design can be applied to an open source native XML database engine, such as eXist-db without altering the underlying indexing scheme. Results obtained show good performance based on the XPathMark benchmark.

The rest of this paper is organised as follows, Section 2
presents the complete system overview and design of the querying and insertion mechanism, while Section 3 discusses the implementation on eXist-db in detail. The results obtained from carrying out these experiments are tabulated in Section 4 and our conclusions are presented in Section 5.

2. SYSTEM OVERVIEW

In this section, we outline our system and describe in detail the proposed cluster based architecture. Section 2.1 describes the system design and introduces all the notions pertaining to the discussion. Section 2.2 illustrates the querying mechanism. Section 2.3 discusses the insertion mechanism employed and analyses the algorithm based on the message complexity.

2.1 System Design

In general, a cluster based approach assigns a cluster head to manage one logically grouped site or cluster. A cluster consists of slave servers which contain the data that conforms to a particular schema or a set of schemas based on the Catalog (defined later), and a client which initiates queries. The cluster head and the slaves work together to respond to the query. Figure 1 illustrates a sample network configuration depicting the various components of our architecture.

The data is assumed to be pre-distributed among the different sites as we have limited to zero influence on the data distribution scheme for any network initially. For this purpose, we introduce the notion of a Catalog. The Catalog tracks the data present at different sites and contains all the coordination data such as configuration, location and schema information of the servers. A cluster head is selected by the Database Administrator (DBA) and this information is also recorded in the Catalog. Also, each cluster has a dedicated server for back up which doubles up as a cluster head in case of failures since fault tolerance [23] is a critical issue in real world scenarios.

In the following section, we present our query processing approach and supporting index structures in the context of pre-distributed XML repositories. For the ease of representation, let \( X = \{x_1, x_2, \ldots\} \) be the set of distinct XML schemas, where \( x_i \) is a distinct XML schema, and \( E = \{e_1, e_2, \ldots\} \) be the collective list of elements in the XML schemas, where \( e_i \) represents an element in the XML schema.

The Catalog maps the schemas to the clusters in which they reside. This can be described as a relation from the set of schemas \( X \), to their respective clusters represented by their cluster heads \( C \).

A Distributed Hash Table (DHT) is maintained at each cluster head. A DHT was chosen for our implementation because it emphasises characteristics of decentralisation, fault tolerance and scalability. This DHT hashes the elements with the servers in which they are present.

2.2 Querying

Query processing begins when the client attaches to a cluster head and fires a query. The incoming query is passed on to the Distributed Query Processor (DQP) of the cluster head (shown in Figure 2). Each cluster head has knowledge about all the other cluster heads and its own slave servers. On receiving a query from the client application, the cluster head refers to its internal DHT for decision making. The Forward XPath Parser of the DQP is responsible for accessing the DHT and returning the list of cluster heads and slave servers which possess the data. The Collator module of the DQP forwards the query to every server in the list; if it is a slave server then a directive to fetch the results is issued. Whereas, if the server is a cluster head, it queries its associated slave servers, if they contain the data. The results from multiple servers are gathered and collated at the Collator which sends the collated result to the querying cluster head and the client in succession.

An efficient distributed query processing algorithm is necessary for controlling the message complexity [14, 3]. In [24], Suciu evaluates the efficiency of an algorithm in a distributed database based on the following criteria: (i) The total number of communication steps is constant, independent of the data or the query. A communication step can be a broadcast, or a gather, and can involve arbitrarily large messages (but see condition ii). (ii) The total amount of data transferred during query evaluation should depend only
on the size of the query’s answer and the total number of cross edges. Based on this, the complexity is theoretically computed as follows: There are two stages in communication between the client and the server — Request phase and Response phase explained below.

**Request Phase**

1. **Client Connection Request**: In this step, one message is required for connection of the client with any one of the cluster heads belonging to $C$. This happens with a probability of $\frac{1}{n}$ where $n = |C|$, the cardinality of set $C$. It denotes the number of clusters in the network.

2. **Client Connection Acknowledgement**: The chosen cluster head, $c_x$, sends an acknowledgment back to the client. This step requires one message.

3. **Client to Cluster Request**: A single message containing the query is sent from the client application to $c_x$.

4. **Cluster to Cluster Request**: Here, the cluster head $s_x$, forwards the query to the appropriate cluster heads in the list obtained from the Forward XPath Parser. The worst case being when all cluster heads contain the requested data thereby sending a maximum of $n - 1$ messages.

5. **Cluster to Slave Request**: At each cluster head the query is sent to the slave servers containing the requested data. To ensure scalability and to reduce the waiting time at each cluster head for data collation, there is a constraint on the number of slave servers managed by a cluster head. Each cluster is limited to have $\log_2 n$ slave servers. In the worst case scenario, i.e., when all of the slave servers have the data, at most $\log_2 n$ messages are sent from cluster heads to their slave servers.

   The initial connection setup between the client and the eXist server takes two messages in total. Therefore, in the worst case, the message complexity in the clusters during the query request phase is 

   $$O(n \log n)$$

**Response Phase**

1. **Slave to Cluster Response**: Each cluster head receives the response from the slave servers that contain the data. In the worst case, again, $\log_2 n$ messages are required. The collator present at the cluster heads collates the queried data and the result is returned as a single message.

2. **Cluster to Cluster Response**: The initiating cluster head, $c_x$ from which the query is directed to other cluster heads receives at most $n - 1$ messages, in the worst case. The resulting data collected from multiple sources is packed into one single message.

3. **Cluster to Client Response**: The collated result obtained in the previous step is returned to the client.

   Thus, the message complexity during the response phase is 

   $$O(n \log n)$$

The total query processing time ($Q_{total}$) is composed of network delay ($N_d$), DHT lookup time ($DHT_{lookup}$), query compilation time ($Q_{compile}$) and query execution time ($Q_{exec}$). Note, here the query compilation time and query execution time is accounted at each of the slave servers. The total query processing time ($Q_{total}$) can be represented as:

$$Q_{total} = N_d + DHT_{lookup} + Q_{compile} + Q_{exec}$$

Query compilation and query execution happen simultaneously at slave servers which have the queried data. These time measures and the DHT lookup time is taken as a constant for all the queries executed for the first time (assuming there is no caching). The only measure which changes is the network delay. The total network delay in this case is composed of the network delay between the cluster heads ($time_{cc}$) and the network delay between the cluster heads and slave servers ($time_{cs}$). The number of requests which are processed at the cluster head is significantly greater than those received at the slave server. This can be represented as:

$$time_{cc} = y \times time_{cs}, \text{ where } y > 1 \text{ is a real number.}$$

From the above explanation, we can deduce that, as the number of cluster heads increase (e.g., increase to 32, 64, etc., cluster heads), the major contributing factor to the query processing time is the network delay.

**2.3 Insertion**

The aim of a distribution design for an XML repository is to formally describe how the XML data is fragmented and allocated at different sites. There are certain correctness criteria to be satisfied in order to ensure that the semantics of the data does not change once the fragmented data is distributed and managed at local sites [10].

Our distribution design models a network where XML data sources are distributed geographically as a result of which initially, the DHT is built based on the information maintained in the Catalog. Usually, it is infeasible to obtain the state of the database servers at the beginning or at one instant of time. Moreover, insertion can happen randomly and the user should have the freedom to upload new files dynamically without the intervention of the DBA. In due course of time, new requirements lead to insertion of large volumes of data and their information has to be maintained to address future queries. As a consequence of which, all the cluster heads require the updated DHT status to guarantee correctness of results for fired queries. Typically, in distributed systems insertion is a recurrent and expensive operation [4]. Thus, a specialized design of bulk insertion is proposed.

A horizontal partitioning method is used for our cluster based architecture. In a cluster, if two or more slave servers contain data adhering to a specific schema then the data is partitioned in a manner where the schemas of the partitioned data are identical.

We define different types of insertion that happens in a distributed database system, namely (i) new data conforming to any existing schema (ii) new data conforming to some new schema. Bulk Insertion can be primarily divided into two phases — Resolve phase and Rehash phase.

In the Resolve phase, the QTable is built, where QTable is a data structure present at every cluster head to store schemas
3. IMPLEMENTATION

Our insertion and query processing modules are incorporated into eXist-db's architecture. All calls to the storage backend are handled by classes, implementing the DBBroker interface. Applications may access a remote eXist server via HTTP messages or XML-RPCs (XML Remote Procedure Calls). In order to support distributed querying, the DQP and the BIM module are added to the XML RPC Interface and XPath Engine. The overview of the extant eXist-db system architecture is shown in Figure 3.

![Figure 3: eXist-db System Architecture](image)

3.1 Distributed Query Processing

The entire distributed query processing is summarised in Algorithm 1. The input is a user provided XPath query. Once the servers are booted, the DHT is built. The initialisation will be carried out as a sequence of operations starting from parsing the Catalog. This will be followed by getting the list of slave servers associated with each cluster head.

Subsequently, a hash map is built which maps each element of the schema present in that cluster to the server URIs (line 2). On the arrival of an XPath query (line 3), the ForwardXPathParser parses it and finds the list of servers to which the query is to be forwarded (line 4). The query is sent to each of the servers in the list and the data is collected and collated (line 6) at the respective cluster head's Collator.

The ForwardXPathParser parses each token of the XPath query (line 9) to determine the URI of the servers which the query is to be forwarded (line 4). The query is sent to each of the servers in the list and the data is collated into eXist-db's architecture. All calls to the storage backend are handled by classes, implementing the DBBroker interface. Applications may access a remote eXist server via HTTP messages or XML-RPCs (XML Remote Procedure Calls). In order to support distributed querying, the DQP and the BIM module are added to the XML RPC Interface and XPath Engine. The overview of the extant eXist-db system architecture is shown in Figure 3.

**Algorithm 1: Distributed Query Processing**

| Data: | userQuery | Data: | serversToBeQueried | Data: | Catalog |
|-------|-----------|-------|---------------------|-------|
| Result: | QueryResult | 1 begin | distributedQueryProcessing |
| | | 2 | DHT ← buildDHT(Catalog) |
| | | 3 if | userQuery then |
| | | | serversToBeQueried ← ForwardXPathParser(userQuery) |
| | | | foreach server in ServersToBeQueried do |
| | | | | QueryResult ← forwardQueryToRemoteServer(userQuery, server) |
| | | | 6 return QueryResult; |
| | | 8 begin | ForwardXPathParser |
| | | | 9 elementList ← split(userQuery) |
| | | | 10 serversToBeQueried ← getServerURIFromDHT(elementList[1]) |
| | | | 11 foreach element in elementList/ i = 2, 3 ... n/ do |
| | | | | serversToBeQueried ← serversToBeQueried ∩ getServerURIFromDHT[element] |
| | | | 12 return serversToBeQueried |

3.2 Bulk Insertion Algorithm

The Resolve and Rehash phases of insertion are detailed in Algorithm 2 and Algorithm 3 respectively.

In the Resolve phase, we find a matching schema for the uploaded file. The uploaded file is parsed to obtain the set of distinct elements (line 3). The XML Element Set Comparator of the BIM returns the list of potential destination servers where the new schema can be uploaded (line 4). The best server is decided based on a similarity measure (line 6–9). XML similarity used in our system can be described as the number of elements in a new XML file which are similar to the existing XML files. The similarity is found out by using WordNet which is a lexical database of English [26]. The new XML file to be inserted is parsed to find the elements present in it. The set of synonyms for each of these elements is found using WordNet. Now at each cluster head the set of synonyms is compared with the set of elements from the new XML file after a lookup on the DHT. A count of the similar elements is obtained. The cluster head with the highest similarity count qualifies to be the target cluster head where the new file is to be inserted. Consequently, an entry in the QTable is made along with the target URI (line 13).

In the Rehash phase, when the bulk insertion job starts, an upload directive is issued for every entry (lines 2–5) in the QTable. Parallelly, another thread serves the requests in the incomingRequestsQueue. Each incoming request in the queue contains the newly uploaded file. The file is parsed to decide whether it is data conforming to some existing
schema or a new schema. In the former case, the upload request is processed and stored at the target slave server. In case of a clash, the server selection is resolved based on pre-assigned probabilities of the servers set by the DBA. An equal probability of \( \frac{1}{d} \) is assigned to each of the slave servers, where \( d \) is the number of slave servers under that cluster head. In the case of a new schema, the new distinct schema elements are added to the DHT and the XML file is uploaded at the target slave server (lines 8–16). The XML Element Set Comparator finds the list of cluster head URIs in which the XML elements are present (line 11). The probability measure used above can be further refined by taking into account various parameters like the network load, processing power of servers and their memory constraints.

4. RESULTS

The experiments were conducted in a test environment consisting of similar machines having the following configuration – Intel i5 processors with a speed of 3.1GHz and 4GB RAM. eXist-db 1.4.0 was installed on a 32-bit version of Ubuntu 10.04. To measure the performance of our distributed system the time interval between the request for the query and the response from the servers in the worst case was measured. This evaluation was done by executing queries on the XMark data set [21] which was chosen since the data set has many structural variations. The XMark DTD represents details of an auction site and was also used for testing in [15, 6, 20].

XML documents adhering to the XMark DTD are uploaded at each of the data servers. A specific set of queries from the XPathMark [9] benchmark is considered for execution on the data. A subset of XPathMark queries executed is shown in Table 1.

Four scenarios were considered for our experiments:

- **Case (i): Centralised setting**
  The experiments were run on the centralised eXist-db without the distribution layer.

- **Case (ii): Distributed setting with two clusters**
  Experiments were conducted on a distributed network which consisted of two cluster heads each managing one slave server (since \( n=2 \) in this case, \( \log_2 n = 1 \)).

- **Case (iii): Distributed setting with four clusters**
  The network consisted of four clusters, each managing two slave servers (\( n=4, \log_2 n = 2 \)).

- **Case (iv): Distributed setting with eight clusters**
  The network consisted of eight clusters, each managing three slave servers (\( n=8, \log_2 n = 3 \)).

All the slave servers were loaded with the different data conforming to the XMark DTD. Twenty thousand files were uploaded in increments of two thousand. The average query execution time for query Q2 (//closed_auction//keyword) over ten trials is tabulated and reported in Table 3. Similarly, the average query execution time for query Q5 (/person//profile//skinname//name) is reported in Table 4. Preliminary experiments were carried out on the dis-
Table 1: XPath Queries

<table>
<thead>
<tr>
<th>Query No.</th>
<th>XPath Query</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1</td>
<td>/site/closed_auctions/closed_auction/annotation/description/text/keyword</td>
</tr>
<tr>
<td>Q2</td>
<td>//closed_auction/keyword</td>
</tr>
<tr>
<td>Q3</td>
<td>/site/closed_auctions/closed_auction/keyword</td>
</tr>
<tr>
<td>Q4</td>
<td>/site/closed_auctions/closed_auction/annotation/description/text/keyword</td>
</tr>
<tr>
<td>Q5</td>
<td>//person/profile/@income]/name</td>
</tr>
</tbody>
</table>

Figure 4: Average Query Execution Time (ms) for Query Q2

Figure 5: Average Query Execution Time (ms) for Query Q5

distributed system and the average query execution time was measured for the aforementioned scenarios. Figure 4 and Figure 5 show the average query execution time for the four defined cases. The number of files in a cluster is denoted on the $x-$ axis and the average query execution time in milliseconds is denoted on the $y-$ axis. The query execution time measured includes indexing time taken by eXist-db’s indexing mechanism. It directly depends on the number of cluster heads present in the network, the number of files present in the clusters and the query itself. As observed from Figure 4, the average query execution time for query Q2 shows an initial increase due to the caching and indexing processes of eXist-db. As the number of files increase, there are no significant anomalies in the graph and the standard deviation was approximately 150ms. In Figure 5, similar results were obtained and the standard deviation was approximately 110ms. From this, we conclude that as the number of files increase, the query execution time steadies and as
the number of cluster heads increase, the query processing time increases due to the network delay.

5. CONCLUSION

In this paper, we have presented a clustered architecture for distributed native XML databases. The implementation focuses on the data structures and algorithms required to support insertion and distributed querying operations. At present, we have addressed the properties of a DHT such as location transparency, scalability and fault tolerance to some extent. This architecture was implemented on eXist-db in a lab setting. The experiments conducted suggest that as the number of cluster heads increase, the query execution time is independent of the number of files and depends mostly on the network delay. The results are encouraging as the total time to query any number of files is observed to be within a specified range. As such a number of avenues for future work are open. Our current implementation is realised only for standard XPath queries, in future we can include support for core functionality queries including XQuery FLWOR expressions and range searches. Another interesting challenge would be distributed query processing when a vertical partitioning approach is adopted over the present architecture. We also intend to take into account congestion/load on the cluster heads and the space requirements on the servers especially on the cluster heads.

6. REFERENCES


