Highly Personalized Information Delivery to Mobile Clients

Bahattin Ozen[1]   Ozgur Kilic[1]   Mehmet Altinel[2]   Asuman Dogac[1]
[1]Software Research and Development Center
Middle East Technical University, 06531, Ankara, Turkey

[2]IBM Almaden Research Center, USA

emails: {turhan,ozgur,asuman}@srdc.metu.edu.tr, maltinel@us.ibm.com

Abstract

The inherent limitations of mobile devices necessitate information to be delivered to mobile clients to be highly personalized according to their profiles. This information may be coming from a variety of resources like Web servers, company intranets, email servers. A critical issue for such systems is scalability, that is, the performance of the system should be in acceptable limits when the number of users increases dramatically. Another important issue is being able to express highly personalized information in the user profiles, which requires a querying power as that of SQL on relational databases. Finally, the results should be customized according to user needs, preferences and the mark up language of their mobile device. Since the queries will be executed on the documents fetched over the Internet, it is natural to expect the documents to be XML documents.

This paper describes an architecture for mobile network operators to deliver highly personalized information from XML resources to mobile clients. To achieve high scalability in this architecture, we index the user profiles rather than the documents because of the excessively large number of profiles expected in the system. In this way all queries that apply to a document at a given time are executed in parallel through a finite state machine (FSM) approach while parsing the document. Furthermore the queries that have the same FSM representation are grouped and only one finite state machine is created for each group which contributes to the excellent performance of the system as demonstrated in the performance evaluation section.

To provide for user friendliness and expressive power, we have developed a graphical user interface that translates the user profiles into XML-QL. XML-QL's querying power and its elaborate CONSTRUCT statement allows the format of the results to be specified. The results to be pushed to the mobile clients are converted to the markup language of the mobile device such as Wireless Markup Language (WML), CHTML or XHTML by the delivery component of the system.

1. Introduction

Mobile access to Internet, made possible through Wireless Application Protocol (WAP) [WAP 00], is increasing dramatically with the new GSM extension called GPRS (General Package Radio Service) [GPRS 00] and therefore serving highly personalized information to mobile clients seems to have a huge market: According to IDC [IDC 00], by 2003, 62 million people will use wireless devices to connect to the Internet and the Strategis Group predicts that 25 million users will want cell-phone access to the services like news and sport headlines, stock quotes, email and online shopping.

Mobile network operators play a major role in delivering the information coming from a variety of resources like Web servers, company intranets, email servers to customers by being strategically positioned between customers and content/service providers. The degree of personalization becomes a key issue in such information services due to limited computation power of mobile devices and overwhelming number of potential users. Currently many Mobile Network Operators (MNO) and software companies are offering some personalized services based on SMS (Short Message Service) that is available in almost all mobile phones. For example Nokia has developed Artus Messaging platform for MNOs, which acts as a gateway between information and applications residing on the Internet or company intranets, and a mobile phone [NA 00]. Messaging platform allows the MNOs to create value-added WAP and messaging applications for all mobile users where users are able to select from the available content links and services the Operator has provided. This allows each user to personalize and control the information they see on their mobile devices. Other systems available in the market today provide similar services; however the level of personalization is limited to choosing from available content links, icons and services.

We believe that the mobile clients will benefit from a much higher level of personalization. For example, a user may wish to receive an immediate alert if within a period of two hours from the start of the trading day, either IBM stock or Microsoft stock is up in at least 3% more than the change in the Dow Jones index. Such complex requests can only be expressed through query languages and none of the systems on the market today provide this level of personalization. In other words expressing highly personalized profiles need a querying power just like SQL provides on relational databases. Since the queries will be executed on the documents fetched over the Internet, it is natural to expect the documents to be in XML [XML98], XML being the emerging standard for data exchange over the Internet. Then the user profiles need to be defined through an XML query language. XML-QL is a good candidate in this respect due to its expressive power as well as its elaborate mechanisms for specifying query results through the CONSTRUCT statement. A point to be noted here is that the users should not be expected to express their profiles through XML-QL but rather a user-friendly interface should be provided to them to automatically create the XML-QL statements. It should be also noted that the mechanisms presented in this paper are applicable to more recently proposed XML query language XQuery as well. Finally, the results to be pushed to mobile clients need to be converted to the markup language of the device.

When such a system providing highly personalized services is deployed on the Internet, the performance becomes a critical issue since the number of users can easily grow dramatically. A key challenge is then to efficiently and quickly process the potentially huge set of user profiles on XML resources. This boils down to developing efficient ways of processing large number of XML-QL queries on XML documents.

Although querying XML documents has been a very active research and development issue recently such as in [FK99, FKMX00, CDTW00], there is no consensus on where to store the XML documents (files, relational databases, object-oriented databases), what should be the corresponding schema and the index structures and how to optimize the queries.

On the other hand, the problem at hand is different from these approaches in the sense that to provide for scalability in such an architecture where the critical issue is the very large number of queries, it makes sense to index the queries rather than documents.

In the work described in this paper, which is being performed as a part of CQMC (Continuous Queries for Mobile Clients) project, such an approach is taken. The users are provided graphical user interfaces to define their profiles from their desktops. Simple profiles can also be defined from mobile devices. These profiles are converted into XML-QL queries. The queries can be change based or timer queries; that is, they need to be activated either when the related XML documents change or the time to execute the query expires. The queries are grouped and indexed such that each element in a query group corresponds to a state in the Finite State Machine (FSM). The system also contains an XML repository. When either there is a change in related XML documents or a timer based query (or a set of queries) needs to be invoked, an event based XML parser is activated that starts sending the events to the Query Execution Engine component of the system that causes the related FSMs to change their states. The Query Execution Engine is capable of capturing the intermediate results during state changes. It should be noted that all the queries that apply to a document are executed in parallel when a document is being parsed and for queries that have the same FSM representation, only one FSM is generated. The results produced are pushed to the related mobile clients.

The rest of the paper is organized as follows: Section 2 briefly summarizes the related work. In Section 3, overall architecture of the system is described. The operation of the system, that is how the query index is created, operation of the finite state machine and the generation of the customized results are explained in Section 4. Section 5 gives the performance evaluation of the system. Section 6 investigates how delta files might be used in the architecture proposed. Finally Section 7 concludes the paper.

2. Related Work

The filtering mechanism described in this paper is influenced by the XFilter system [AF 00]. XFilter is designed and implemented for pushing XML documents to users according to their profiles expressed in XML Path Language (XPath) [CD99]. It takes the advantage of embedded schema information in the XML documents to create better user profiles compared to existing keyword based systems. While doing that, it provides efficient filtering of XML documents with the help of profile index structures in its filter engine. XFilter converts each XPath query into a Finite State Machine (FSM) to deal with XPath structures effectively. However as the name implies, Xfilter is a filtering

mechanism; it does not execute the XPath queries to produce partial results. Therefore when a document matches a user's profile, the whole document is pushed to the user. This feature prevents XFilter to be used in mobile environments since the limited capacity of the mobile devices is not enough to handle or process the entire document let alone to receive it.

Furthermore, XFilter does not exploit the commonalities between the queries, i.e. it generates a FSM per query. This observation motivated us to develop mechanisms that uses only a single FSM for the queries which have common element structure. As demonstrated in Section 5, this improvement boosted the system performance drastically. Also the profiles may involve complex queries requiring the use of XML-QL, which has more expressive power compared to XPath. In providing customized results to the mobile clients, the result construction features of XML-QL also help. Finally, timer based queries are an integral part of such systems and this feature is also not available in XFilter.

Another related work is NiagaraCQ system [CDTW00], which uses XML-QL to express user profiles. It provides measures of scalability through query groups and cashing techniques. However, its query grouping capability is based on execution plans which is completely different from our approach and the performance results reported in [CDTW 00], that is, the execution times of queries, do not make such an architecture a possible candidate for mobile environments. Similar to NiagaraCQ, we also replace constants in a query with parameters to be able to create syntactically equivalent queries, which lead to the use of the same FSM for them.

A survey of continuous queries over data streams is presented in [BW 01] which further indicates the research directions in this area.
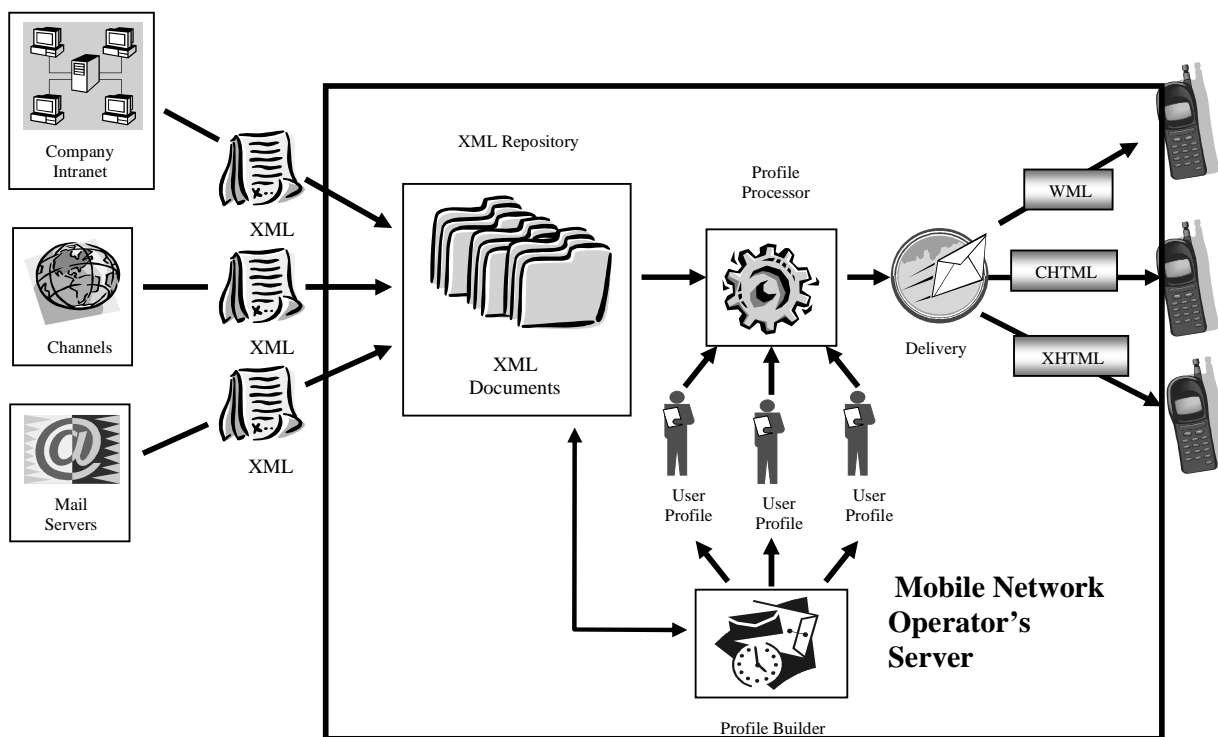
3. Overall Architecture of the System

Figure 1. Overall Architecture of the System

The overall architecture of the system is depicted in Figure 1. XML repository contains the XML DTDs and the corresponding data files. Profile Builder component of the system allows for visual query defining capabilities and also manages the user profiles. Profile Processor first creates query indices for user profiles and then parses the documents to obtain the query results. Delivery component of the system pushes the results to the related mobile clients. As indicated in Figure 1, these major system components reside in the server of the Mobile Network Operator (MNO), which is positioned between customers and content/service providers and plays a major role in delivering the information coming from a variety of resources to customers. Among the system components,

Profile Builder is a Web application and hence the browser part of this software resides at the end user device to help with the user to define his profile.

The overall architecture of the system is depicted in Figure 1. XML repository contains the XML DTDs and the corresponding data files. Profile Builder component of the system allows for visual query defining capabilities and also manages the user profiles. Profile Processor first creates query indices for user profiles and then parses the documents to obtain the query results. Delivery component of the system pushes the results to the related mobile clients. As indicated in Figure 1, these major system components reside in the server of the Mobile Network Operator (MNO), which is positioned between customers and content/service providers and plays a major role in delivering the information coming from a variety of resources to customers. Among the system components, Profile Builder is a client/server application and hence a part of this software resides at the mobile device to help with the user to define his profile.

The main components of the system are explained briefly in the following:

a. XML Repository: XML data coming from diverse data resources like channels (news, entertainment, stock prices, etc.), email, Web, file servers are stored as files in an XML repository. The main functionality of XML repository manager is to inform the Profile Processor when it receives a new XML DTD or an XML file.

The data sources could be pull based or push based. Push based data sources inform the repository whenever interesting data is changed. On the other hand the repository manager checks changes on pull-based data sources periodically.

b. Profile Builder: Profile Builder provides facilities through which a user can see his current queries in a list and manage them by adding new queries, activating/deactivating a query defining his profile.

The first phase of a profile creation process is resource selection. Users select the resource XML documents, stored in the repository, on which they want their queries to be executed. In the resource selection part, the DTDs available in the system and XML files confirming to these DTDs are depicted to the user. The resource selection screen is generated dynamically by parsing an XML document that contains the information about available XML documents in the repository grouped by their DTD's. The resource selection screen for an example repository is given in Figure 2. Note that a user may define any number of queries to reflect their profiles.

The resource selection screen lets users to choose the XML documents on which they want their queries to run. The user is able to select a category (i.e. DTD), or a set of XML files. If a category is selected then the XML-QL query is executed on all the XML documents conforming to the selected DTD. The result of the query is the union of the results obtained from each of the XML file(s). The second screen of the Profile Builder, which is dynamically generated according to DTDs, allows a user to create or update a profile based on a selected DTD or selected XML file(s) as shown in Figure 3.
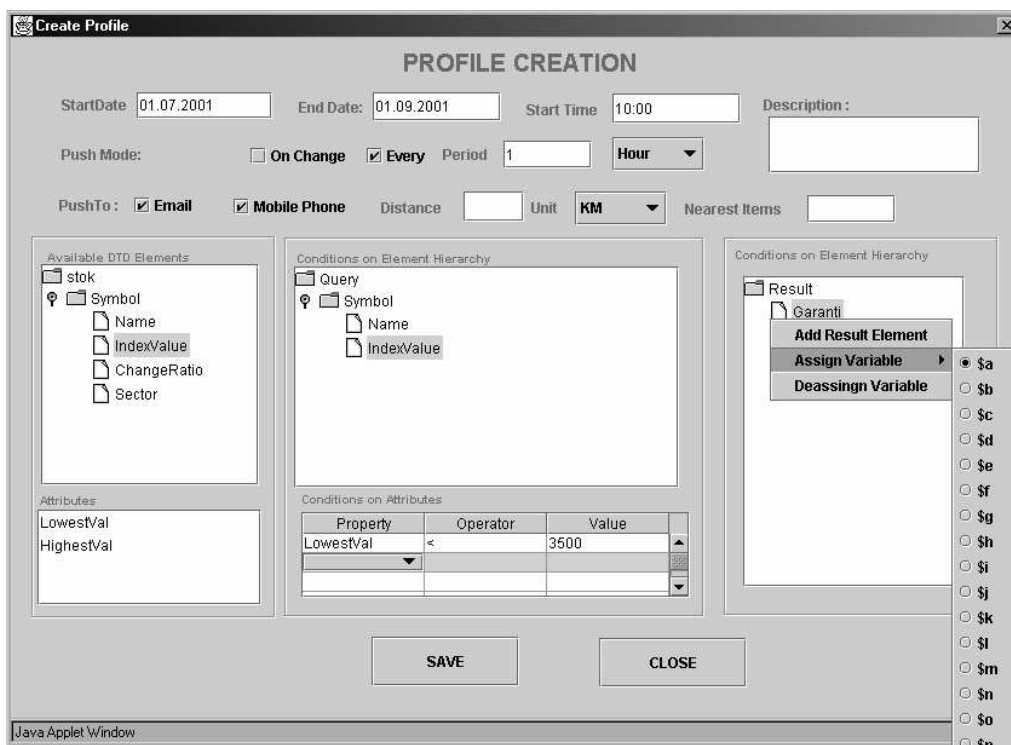

Figure 2. Source Selection Screen

Figure 3. Visual Profile Builder

Queries in a user profile can be classified into two categories depending on the criteria used to trigger their execution. Change based queries are fired as soon as new relevant data becomes available. Timer-based queries are executed only at time intervals specified by the user. The query becomes effective at the start time. The time interval indicates how often the query is to be executed. Queries are deleted from the system automatically after their expiration time. "Push mode" specifies both execution mode of the query and where to send the query result which could be an email address or a mobile phone number. Profiles defined through Visual Profile Builder are transformed into XML documents which contain XML-QL queries as shown in Figure 4.

```
<Profile>
  <XML-QL>
              WHERE   <Symbol> <Name >GARAN</Name>
                                <IndexValue LowestVal=$b >$a</IndexValue>
                      </Symbol>
                      $b < 3500
              IN "imkb.xml"
              CONSTRUCT   <Result>
                                  <Garanti>$a</Garanti>
                          </Result>
  </XML-QL>
  <StartDate>  01.07.2001 </StartDate>
  <StartTime>  10:00     </StartTime>
  <EndDate>    01.09.2001 </EndDate>
  <PushMode onChange="no">
    <Every>  <PeriodSize>1</PeriodSize> <PeriodType>Hour</PeriodType>  </Every>
      <PushTo email="on" mobile="on"/>
   </PushMode>
</Profile>
```
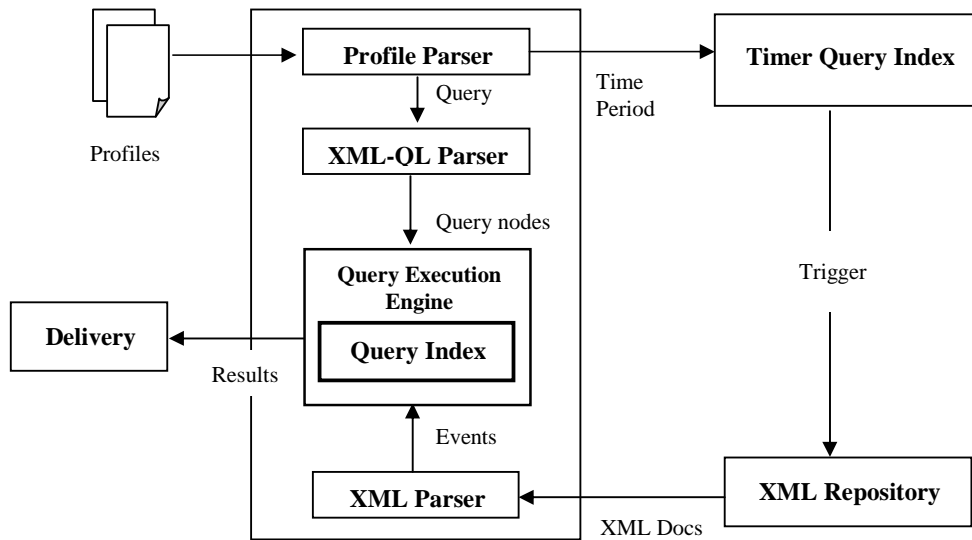
Figure 4. Profile Syntax represented in XML (Grey area shows the XML-QL query)

Figure 5. Profile Processor

c. Profile Processor: The basic components of the Profile Processor shown in Figure 5 are as follows: 1) an event-based XML parser, which is implemented using SAX API [Meg98], for XML documents; 2) a profile parser that has an XML-QL parser for user profiles and creates the Query Index; 3) a Query Execution Engine which contains the Query Index which is associated with Finite State Machines to query the XML documents; and 4) a dissemination component that pushes the results to the related users. When a document arrives at the Profile Processor, it is run through an XML Parser that then drives the process of query execution through the query index. The results to be pushed to the mobile clients are converted to the markup language of the mobile device such as CHTML, XHTM, or WML, whereas for the results to be sent to the email addresses a pre-defined style sheet is used.

d. Delivery Component: Delivery component of the system pushes the results to the related users.

4. Operation of the System

The system operates as follows: Profile Builder informs Profile Processor when a new profile is created or updated; the profiles are stored in an XML file that contains XML-QL queries, execution conditions (time-base or change-base), and addresses to dispatch the results (see Figure 4). The Profile Parser component of the Profile Processor parses the profiles; XML-QL queries in the profile are parsed by an XML-QL parser. While parsing the queries, the XML-QL parser creates FSM representation of each query, if the query does not match to any existing query group. Otherwise, the FSM of the corresponding query group is used for the input query. FSM representation contains state nodes for each element name in the queries which are stored in the Query Index.

At the profile parsing time, another data structure called *Timer Query Index* is created which contains a sorted list of time-based queries and a set of pointers to the related XML documents. When the time expires for those queries in the *Timer Query Index*, the Profile Processor is alerted to parse the related XML document. Similarly when a new document arrives, the XML Repository Manager alerts the Profile Processor so that the related XML document is parsed. The event based XML parser sends the events to the Query Execution Engine. The handlers in the Query Execution Engine respond to these events. By moving the FSMs to their next states when current states succeed certain checks like evaluating the attributes, level checking or pattern matching for character data. In the mean time the data in the document that matches the variables are kept in content lists so that when the FSM reaches its final state, all the necessary partial data to produce the results are ready to be formatted and pushed to the related mobile clients. Note that all the queries that are to be executed on an XML document at a given time are executed in parallel in one parsing of the document.

4.1. Creating Query Index

The state changes of a FSM are handled through the two lists associated with each node in the Query Index (See Figure 8): The current nodes of each query are placed on the Candidate List (CL) of the index entry for its corresponding element name. All of the query nodes representing future states are stored in the Wait Lists (WL) of their corresponding element name. Copying a query node from WL to the CL represents a state transition in the FSM. Notice that the node copied to the CL also remains in the WL so that it can be reused by the FSM in future executions of the query since the same element name may reappear in another level in the XML document.

It should be noted that this system is developed to handle very large number of queries and in such a set it is quite probable that there will be queries that have the same tree structure and the same element names, that is, the same FSM representation but different constant values. In this case a single FSM can handle these queries and as demonstrated in Section 5 this greatly enhances the performance of the system.

When the query index is initialized, the first node of each query tree is placed on the CL of the index entry for its respective element name. The remaining elements in the query tree are placed in respective WLs. Query nodes in the CL indicate that the state of the query might change when the XML parser processes the respective elements of these nodes. When the XML parser catches a start element tag and if a node in the CL of the element in the Query Index satisfies level check and attribute check as explained in Section 4.2, and then the nodes of the immediate child elements of this node in the Query Index are copied from WL to CL. The purpose of the level check is to make sure that the element appears in the document at a level that matches the level expected by the query. The attribute check applies any simple expressions that reference the attributes of the element.

Consider an example XML document and its DTD given in Figure 6 and the example queries and their FSM representations given in Figure 7. Note that there is a node in the FSM representation corresponding to each element in the query and the FSM representation's tree structure follows from XML-QL query structure.

```
<!ELEMENT disk (label, contents)>
<!ELEMENT contents (directory*, file*)>
<!ELEMENT directory (name, size, type, contents)>
<!ELEMENT file (name, size, type)>

<disk>
  <label> C </label>
  <contents>
    <directory>
              |         <name> Java Projects </name>  <size> 1 MB </size>
                        <contents>
    Level: 4            <directory> <name> Sources </name>  <size> 5 KB </size>
                 |              <contents>
                                  <file>
             Level: 6             <name>MySwing.java</name> <size> 5 KB </size> <type>JAVA</type>
                                  </file>
                                </contents>
                        </directory>
                        <file> <name>Main.exe</name> <size> 5 KB </size> <type>EXE</type> </file>
                    </contents>
        </directory>
        . . . .
  </contents>
</disk>
```
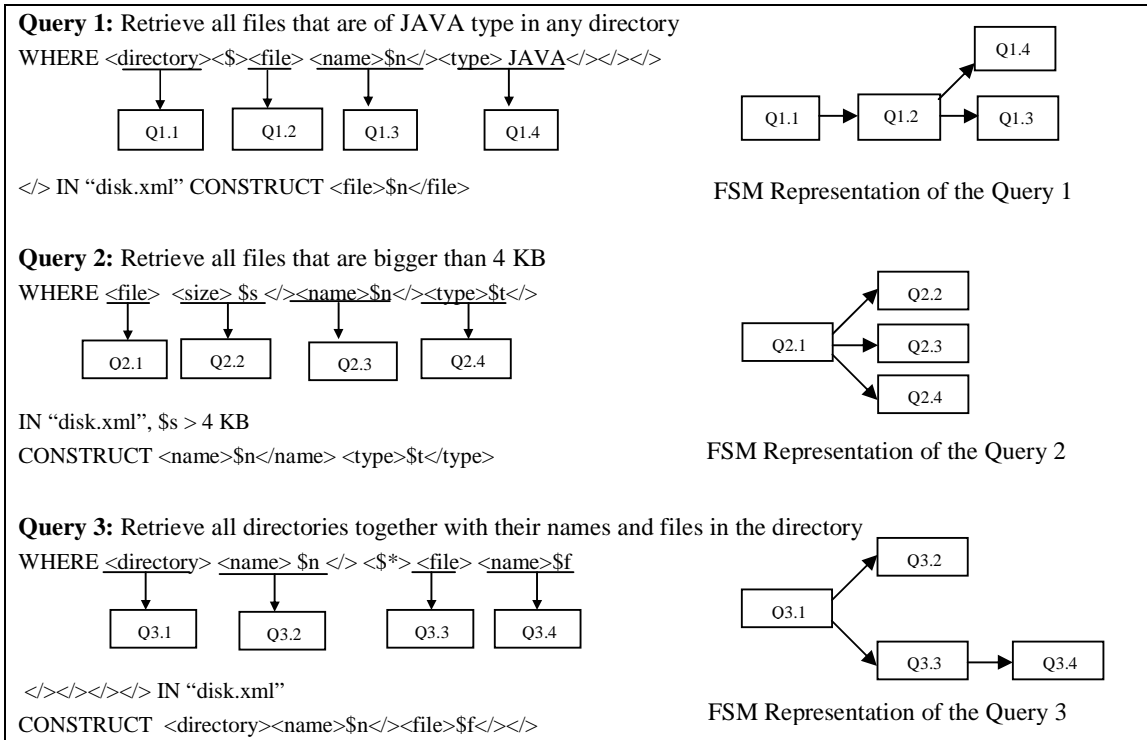
Figure 6. An Example XML Document and its DTD (disk.xml)

Query 1: Retrieve all files that are of JAVA type in any directory
WHERE <directory><$><file> <name>$n</><type> JAVA</></></>

Q1.1	Q1.2	Q1.3	Q1.4

</> IN "disk.xml" CONSTRUCT <file>$n</file>

FSM Representation of the Query 1

Query 2: Retrieve all files that are bigger than 4 KB
WHERE <file> <size> $s </><name>$n</><type>$t</>

O2.1	Q2.2	Q2.3	Q2.4

IN "disk.xml", $s > 4 KB
CONSTRUCT <name>$n</name> <type>$t</type>

FSM Representation of the Query 2

Query 3: Retrieve all directories together with their names and files in the directory
WHERE <directory> <name> $n </> <$*> <file> <name>$f

Q3.1	Q3.2	Q3.3	Q3.4

</></></></> IN "disk.xml"
CONSTRUCT <directory><name>$n</><file>$f</></>

FSM Representation of the Query 3

Figure 7. Example Queries

The structure of the query index for the example queries as well as the structure of query nodes is given in Figure 8.

(a) Query Index

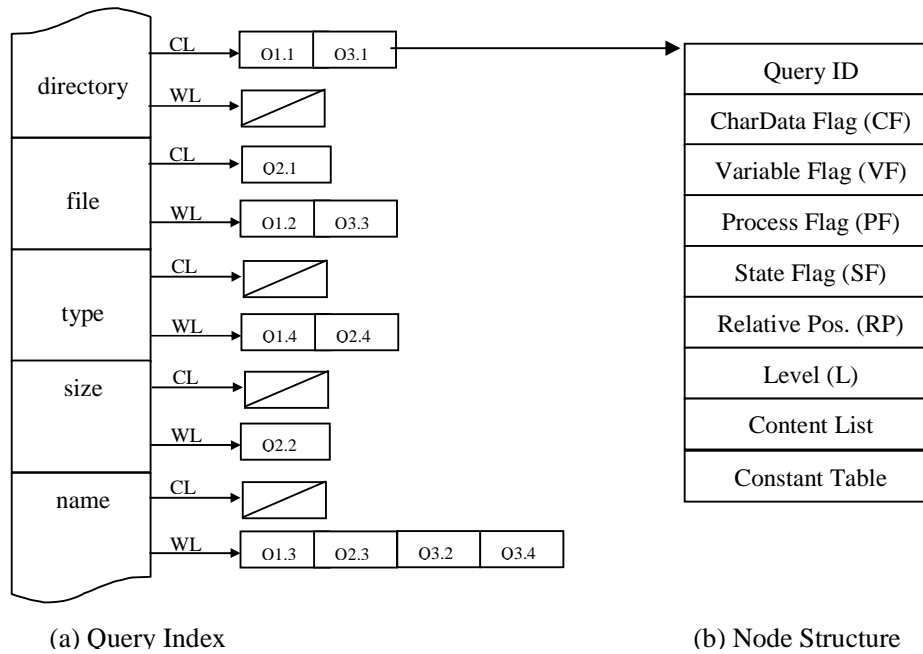| |
|---|
| Query ID |
| CharData Flag (CF) |
| Variable Flag (VF) |
| Process Flag (PF) |
| State Flag (SF) |
| Relative Pos. (RP) |
| Level (L) |
| Content List |
| Constant Table |

(b) Node Structure

Figure 8. Initial states of the Query Index for example queries

Each node in each query has a unique identifier. Other elements of the node structure are as follows:

- *CharData Flag* is on when the node in question has a character pattern to be matched,
- *Variable Flag* is on for nodes that have variables,
- *Process Flag* is used for dual purpose; it is set to true for nodes containing character pattern when the start element tag is encountered so that the pattern can be matched in the element data handler. For nodes containing variables, it is set to true when the start element tag is encountered so that the content of the variable can be generated.
- *Level* is an integer that represents the level in the XML document at which this query node should be checked. Because XML does not restrict element types from appearing at multiple levels of a document and because XML-QL allows queries to be specified using "relative" addressing in addition to "absolute" addressing, it is not always possible to assign this value during query parsing. Therefore, this information needs to be updated during the execution of the query.
- *State flag* is set to true when the element of this node is successfully processed that is if the level check, the attribute check and/or data comparison are satisfied.
- *Relative position* is an integer that describes the distance between this query node and the previous (in terms of position) query node in a query tree.
- A *Content List* stores intermediate results for a variable. After finishing XML parsing, outputs are generated from these lists. It should be noted that nested elements can appear in an XML document and therefore generated contents of a variable can occur at different levels. Hence there is a need for another *level* mechanism to distinguish variable contents for elements at different levels. Figure 9 shows the data structure of the content list. A node in the list has a level, content buffer, and a boolean flag called active flag. When start element event is generated, an empty content node is inserted into the content lists of variable nodes and the active flag is set to on. Then, XML data emitted with events is appended into all active contents. In end element event handler, the flag is again set to off position to end content accumulation to the active contents. There is a table in the query execution engine called *Active Content Table*, which keeps tracks of active nodes and provides one shut access while appending data to the active nodes.
- For queries having the same tree structure with same element names but with different constants, only one FSM representation is generated. *Constant Table* is used for such queries to hold the values of different constants as shown in Figure 10.
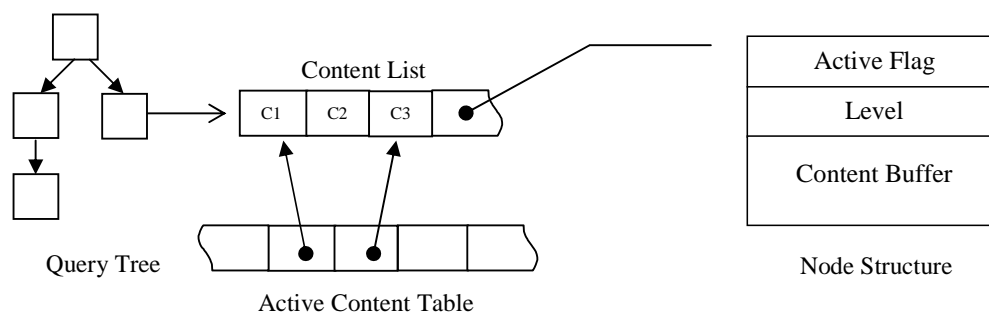

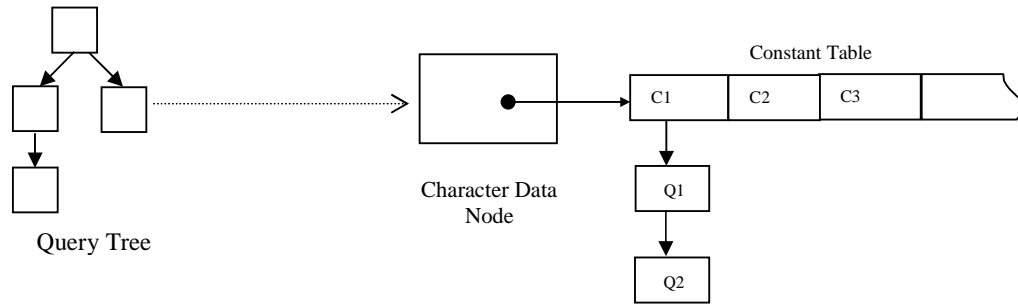Figure 9. Content List and Its node structure

Figure 10. Constant Table

4.2 Operation of the Finite State Machine

When a timer event or a new XML document activates the XML SAX parser, it starts generating events. The following event handlers handle these events:

Start Element Handler checks whether the query element matches the element in the document. For this purpose it performs a level and an attribute check. If these are satisfied, depending on the type of the query node it either enables data comparison or starts variable content generation. As the next step, the nodes in the WL that are the immediate successors of this node are copied to CL at this stage. Before the copy operation, the level value of the new node is calculated with the current level value of the document and relative position value of the current node. Even in a single document, the FSM may be executed more than once if the same element names reappear in the document. Therefore there is a need to reinitialize the FSM. Furthermore XML documents can be nested, that is, the same element may appear at different levels (consider directory in Figure 6). Therefore it may be necessary to generate a FSM to handle this recursion. This is achieved by copying this new node to CL in the query index.

End Element Handler evaluates the state of a node by considering the states of its successor nodes and when the root node is reached it generates the output. End element handler also deletes the nodes from CL which are inserted in the start element handler of the node. This provides "backtracking" in the FSM.

Element Data Handler is implemented for data comparison in the query. If the expression is true, the state of the node is set to true (in state flag) and this value is used by the End Element Handler of the current element node.

**End Document Handler** signals the end of result generation and passes the results to the Delivery Component.

A detailed algorithm of the query execution process is presented in the Appendix I. In this section, we will highlight important steps of the algorithm through an example.

For the document given in Figure 6 and the queries given in the Figure 7 the processing proceeds as follows; the start element tag encountered, *<directory>* at level 4 in the document, causes the state machine for Q1 and Q3 to progress to their next states. Consequently, node Q1.2 and node Q3.3 are copied to CL list of *<file>* element and node Q3.2 is copied to CL list of *<name>* element as shown in Figure 11.a. After the *<directory>* tag, next element in the document is start element tag of *<name>* and Q3.2 is the only node in the CL of *<name>* element. Since Q3.2 contains variable $n, an empty node is inserted into the content list of Q3.2 as shown in Figure 11.b. Then, XML parser reads the character data which happens to be "Java Projects". The Query Execution Engine continues to process the nodes in the CL of *<name>* element. Since Q3.2 is the only node in CL and it contains variable, no pattern matching is done. In this state, character data read is written into the Content Lists of the variable nodes. Afterwards, the level info is updated. Figure 11.c shows the states of the Query Index after processing end element tag *</name>*.

As the parser reports events, the states of queries change as given in the Algorithm Query Execution Process (see Appendix I). When the XML parser sends the event for the start element *<directory>* at level 6 in the document, the Query Index realizes the nested instances of the same element, namely *<directory>*. As given in the algorithm, it puts a copy of Q1.1 and Q3.1 into CL of *<directory>* and updates level info for them. Q1.1° and Q3.1° are the copied nodes shown in Figure 11.d. This copying is necessary to handle multiple nested occurrences of *<directory>*. In this way, more than one FSM are run for a query in the XML document. Also, as given in the algorithm, next nodes of these nodes are copied and are put into corresponding CLs with updated level information.

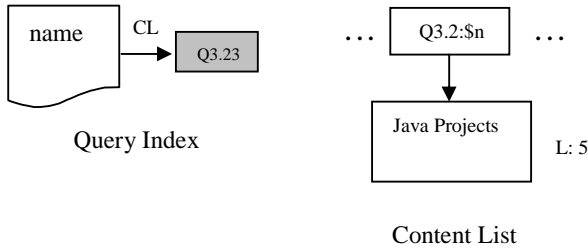After processing the start element *<directory>*, at level 6, the current states of Query Index are depicted in Figure 11.d.

The next two figures, Figure 11 (e) and (f) show CL of *<name>* in the Query Index and content list of Q3.2:$n in the Content list after processing start element tag *<name>* and after processing end element tag *</name>* respectively. Notice that the variable $n of Q3 is partially generated at this step. When the end element tag *</directory>* is encountered and query is satisfied, deactivated content nodes in the content lists of the variables are written into the partial results.
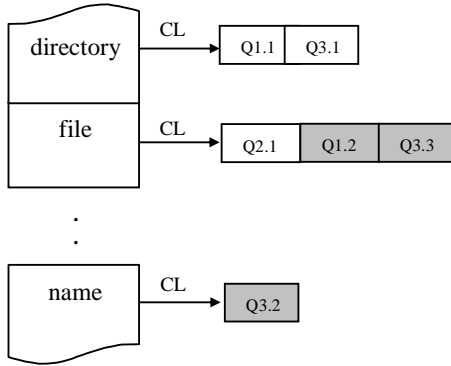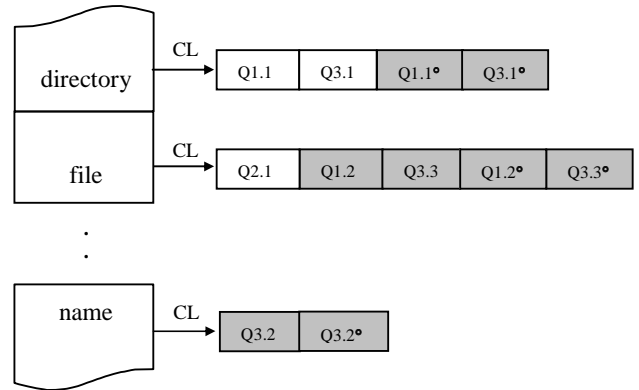

a) After processing *<directory>* at level 4
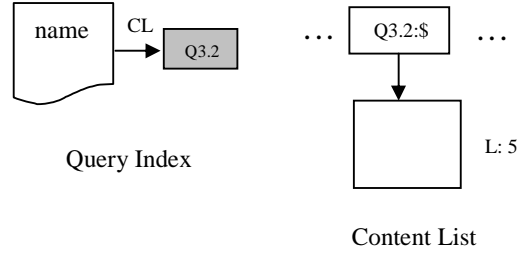
b) After processing *<name>* at level 5

c) After processing *</name>* at level 5

d) After processing *<directory>* at level 6
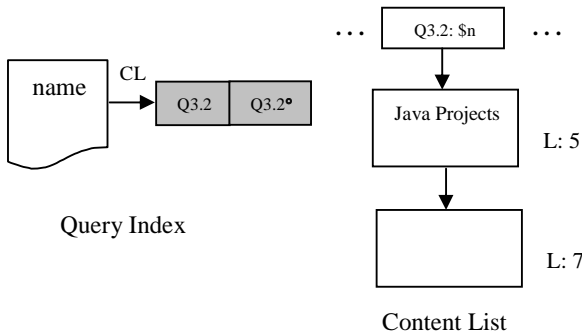
e) After processing *<name>* at level 7
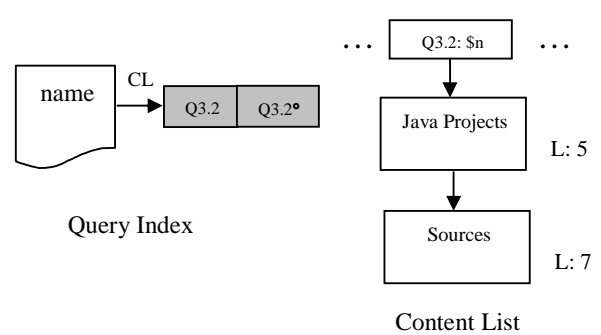
f) After processing *</name>* at level 7

Figure 11. The states of the Query Index and Content list during the execution

4.3 Generating Customized Results

Results are generated when the end tag element of the query's root element is encountered. Content lists of the variable nodes are traversed to fetch content groups. These content groups are further processed to generate results. This process is repeated until the end of the document is reached.

The results need to be formatted as defined in the CONSTRUCT clause. Therefore while parsing the document to generate the results conforming to the structure defined in the CONSTRUCT clause not only a query tree but also a Construct Tree is generated. An example query and its corresponding Construct tree is depicted in Figure 12.

```
WHERE    <book> <title>$t</>
                <author>$a</>
                <publisher>
                <name>ADDISON WESLEY</>
                </>
          </> IN "bib.xml"
CONSTRUCT  <result>
                    <title>$t</>
                    <author>$a</>
                </>
```
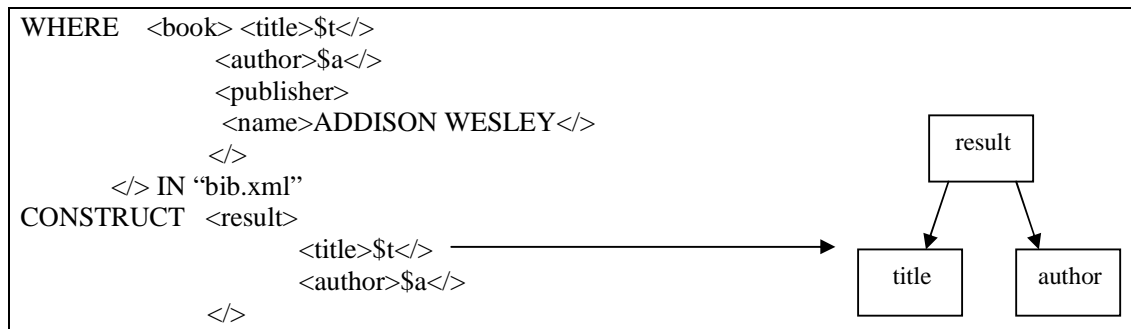
Figure 12. An example query and its corresponding Construct tree

CONSTRUCT part of the query tells how the result sets should be generated and gives the hierarchy of the resulting XML document. We keep this hierarchy in the Construct tree structure. When the end element of the root node is encountered by the parser, it checks whether a result element can be generated by considering the state of the root node of each query. For the satisfying results, construct tree is used for output generation in such a way that collected contents of the variables are merged and the output elements are formed. In Figure 14 this generation is shown for the example query. For each title name in the content list of query node title, an author is selected from the content list of the query node author and a result element is formed.

It should be noted that CONSTRUCT clause in XML-QL may also contain WHERE clauses that query the collected content of the original query, an example of which is given in Figure 13. In this query, the WHERE part of the query when executed produces content for $t and $p variables (see Figure 15). The nested WHERE clause is executed on the content collected for $p to collect content for $a for each $t value. In our architecture this nested query is executed by also forming a query index. Note that there can be several WHERE clauses in the CONSTRUCT clause, each of which recursively uses the collected query content of the former query. When it comes to generating the results, the result of the inner most query is merged with the immediate outer query contents in the Construct tree.

```
WHERE    <book> <title>$t</>
                <publisher><name>ADDISON WESLEY</>
                </>
          </> CONTENT_AS $p IN "bib.xml"
CONSTRUCT  <result>
                        <title>$t</>
                        WHERE  <author>$a</>IN $p
                        CONSTRUCT <author> $a</>
                </>
```
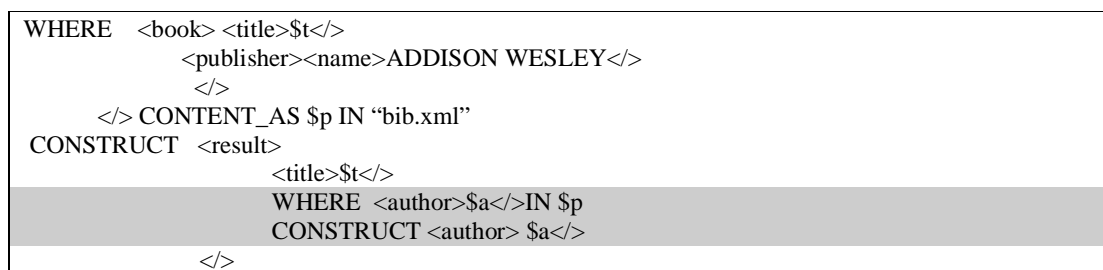
Figure 13. A Sample Nested Query

Figure 14. Generated result for the example query
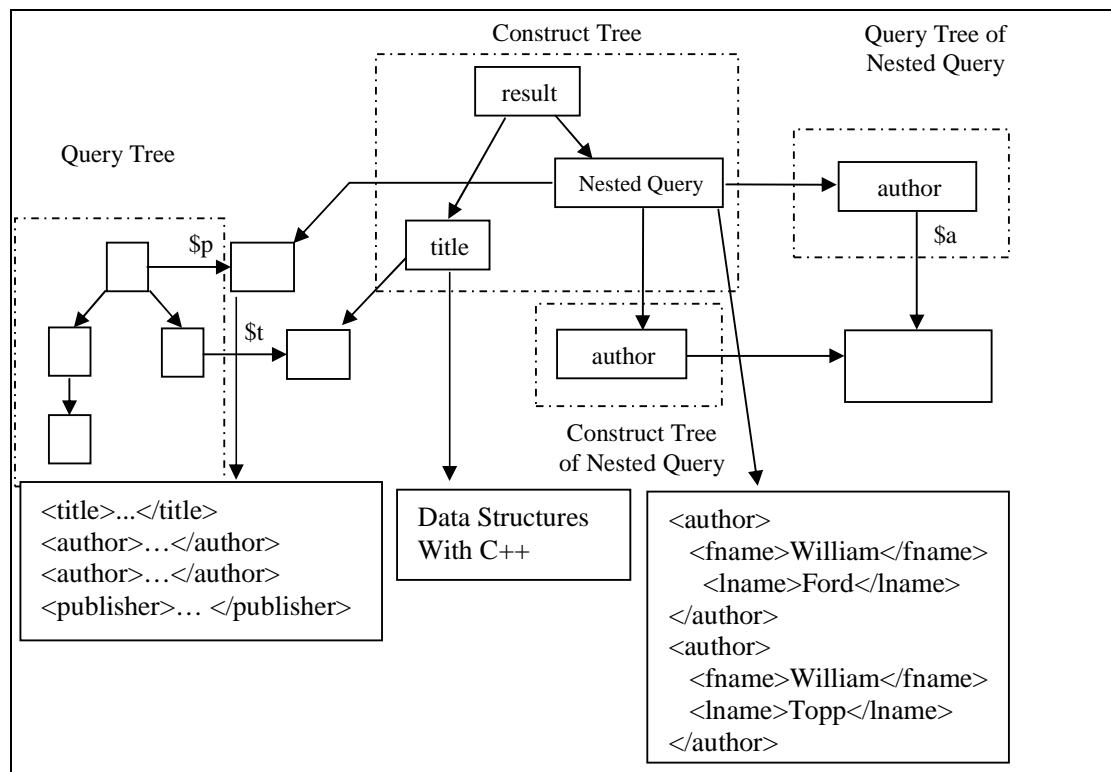
Figure 15. Result Generation for nested queries

4.4 Delivery of the Results

At the end of the XML parsing the result of the each query is written to a file. The Delivery Component gets Ids of queries whose results are ready for delivery operation from the ID Matcher table. For each query whose result is ready for delivery, the result and the XSL [XSL 00] files of the

13

result document are acquired. ID Matcher table gives only the file names of the profiles. Since the generated result files have the same name with the profile but have a different extension ("rst"), Delivery Component finds corresponding XSL and output filename according to the file name of the profiles. Delivery component also obtains the email address and the mobile phone number of the user from the account information document.

XSL files are used to format the generated result document. These XSL files are constructed during the profile generation step as mentioned in previous sections. Basically, XSL files specify the rules to convert the resulting documents to relevant markup language. Figure 16 shows the delivery component and its interactions.
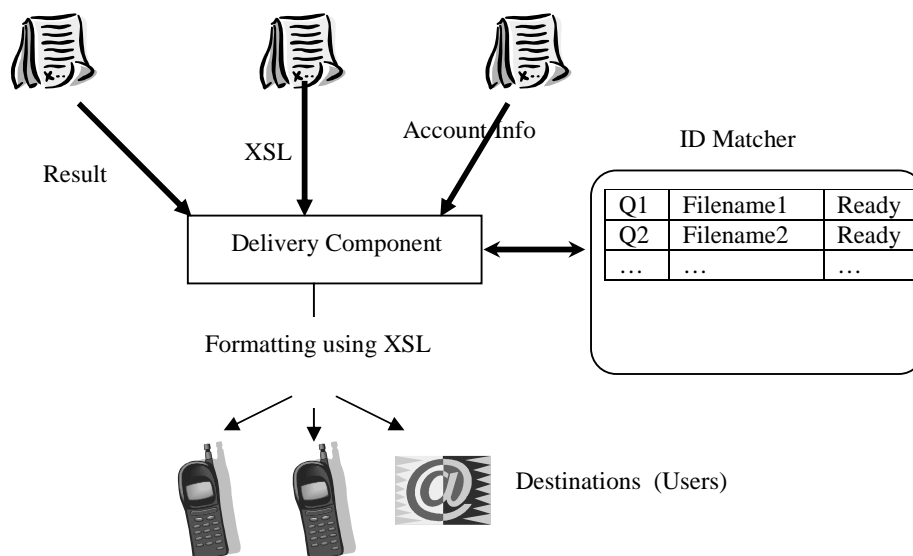
Figure 16. Delivering Results to Users (delivery component and its interactions)

The results of the queries that to be sent to mobile devices are converted into the markup language of the mobile device such as CHTML [CHTML], XHTML [XHTML] and WML [WML 00]. Example style sheets for presenting the result in CHTML and WML are given in Figures 17 and 18. How the results appear in mobile devices is shown in Figure 19. For results that will be sent to e-mail addresses, a pre-defined XSL query is used to generate e-mail messages.

```
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
<xsl:output method="chtml"/>
<xsl:template match="/">
<html>
      <head>
         <META http-equiv="Content-Type" content="text/html; charset=utf-8">
         <META name="CHTML" content="yes">
         <META name="description" content="result document">
         <title> <xsl:value-of select="/results/description"/> </title>
      </head>
      <body>
      <center><b><xsl:value-of select="/results/description"/></b></center>
            <hr>
            <I>The results of your query are:</I><br>
            <xsl:for-each select="/results/result">
                    <b><xsl:number value="position()" format="1"/>.</b>
                    <xsl:value-of select="./drugstore/name"/>;
                    <xsl:value-of select="./drugstore/address"/><BR>
            </xsl:for-each>
      </body> </html> </xsl:template> </xsl:stylesheet>
```
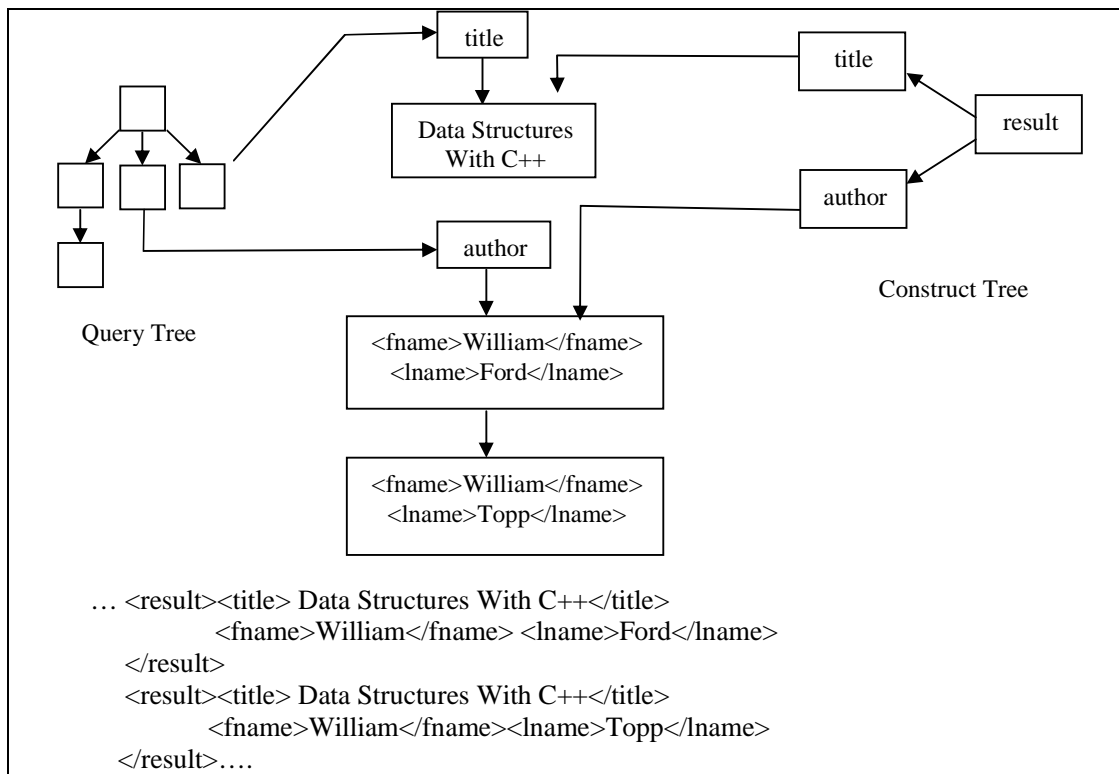
Figure 17. Style sheet for CHTML

```
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
<xsl:output method="wml"/>
<xsl:template match="/">
<wml>
        <card id="start" title="The results of your query are:">
                <do type="prev" label="Back"><prev />
                </do>              <p>
                      <table columns="2" align="LCC">
                              <tr><td>No:</td><td>Result</td></tr>
                              <xsl:for-each select="/results/result">
                              <tr>
                                  <td><xsl:number value="position()" format="1"/></td>
                                  <td><xsl:value-of select="./drugstore/name"/>;
                                        <xsl:value-of  select="./drugstore/address"/></td>
                              </tr> </xsl:for-each>
                      </table> </p> </card>
</wml> </xsl:template> </xsl:stylesheet>
```
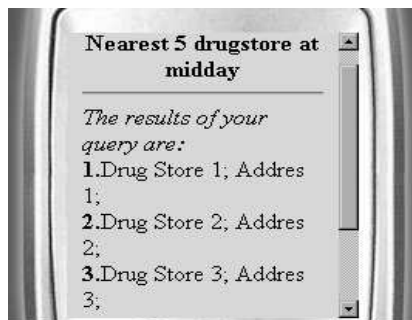
Figure 18. Style sheet for WML

(a) CHTML

(b) WML

Figure 19. Some Presentation Samples for Query Results in Mobile Devices

5. Performance Evaluation of the System

The system is implemented using MS Visual C++ version 6.0. Implementation details can be found in [Kilic 01, Ozen 01]. In this section we present the results of the performance evaluation of the system.

The experiments are conducted on a Pentium III 650 MHz PC with 128 MB memory running MS Windows 2000. All structures are kept in memory (i.e. there is no disk I/O). The characteristics of XML documents used in the tests are given in Table 1. Types of sample queries used in the experiments are given in Figure 20.

Before we start presenting the results of the experiments we find it important to note the following:

1. Our experiment environment (i.e., the Pentium III PC and 128 MB memory) is a modest one. CQMC project is being developed for mobile network operators and obviously, in an industrial environment with a powerful server and a large main memory, the performance of the system will improve.
2. Furthermore the architecture allows to distribute the processing to more than one server very easily.
3. Finally, the performance of the system can further be enhanced by using only the changed data (delta files) rather than the original data file which is investigated in Section 6.

15

Therefore although the results are presented for a maximum of 100,000 queries (due to memory limitations), we expect that the performance of the system will still be acceptable for mobile environments for millions of queries since the results of the experiments show that the system is highly scalable.

Table 1. Test Data

File	Size	Depth	Number of Elements	Parsing time
Bib1.xml	1 KB	4	35 elements, 7 attributes, 291 characters	1-2 ms
Bib2.xml	50 KB	4	1733 elements, 445 attributes, 15317 characters	40 ms
Bib3.xml	500 KB	4	17475 elements, 4368 attributes, 137141 characters	592 ms
Bib4.xml	1 MB	4	40321 elements, 10080 attributes, 356742 characters	1104 ms

```
WHERE <book> <title>The New New Thing</>
              <author>$a</>
        </>      IN "bib.xml"
CONSTRUCT <author>$a</author>

WHERE <name atr1="1" atr2="2">$a</name>
        IN "bib.xml"
CONSTRUCT <name>$a</name>

WHERE <book><title> $t</>
            <author>
                <firstname>$f</>
                <lastname>$l</>
          </>
        </> CONTENT_AS $p IN "bib.xml"
CONSTRUCT <result><title> $t</>
            WHERE <authors> IN $p
                <firstname>$f</>
                <lastname>$l</>
            </authors>

WHERE <book>
        <title>$t</>
        <author>
            <firstname>$f</>
            <lastname>$l</>
        </>
        <publisher><name> Addison-Wesley</></>
      </> IN "bib.xml"
CONSTRUCT <result>
            <title> $ </>
            <authors>
                    <firstname>$f</>
                    <lastname>$f</>
            </>
        </>
```

Figure 20. Sample Queries

 We conducted three sets of experiments to demonstrate the performance of the architecture for different document sizes and query workloads.

5.1 Scalability Experiments

The first set of experiments are performed by varying the number of query groups where the test data given Table 1 is used and the number of queries ranged from 50 to 100,000. The graph shown in Figure 21 contains the results for different query groups, that is, the queries have the same FSM representation but different constants, for the document Bib1.xml (1KB). This graph also shows the results of an experiment where the query groups are not taken into consideration. These experiments indicate that proposed architecture is highly scalable and a very important factor on the performance is the number of query groups and that generating a single FSM per query group rather than per query is well justified. It is clear that when you have very large number of queries on the same XML document, many queries will tend to be similar. Consider for example, the users requesting stock quotes; such queries will all have the same FSM representation where only a constant of a condition, namely, the company name will change. And the CQMC system is designed to handle query groups.
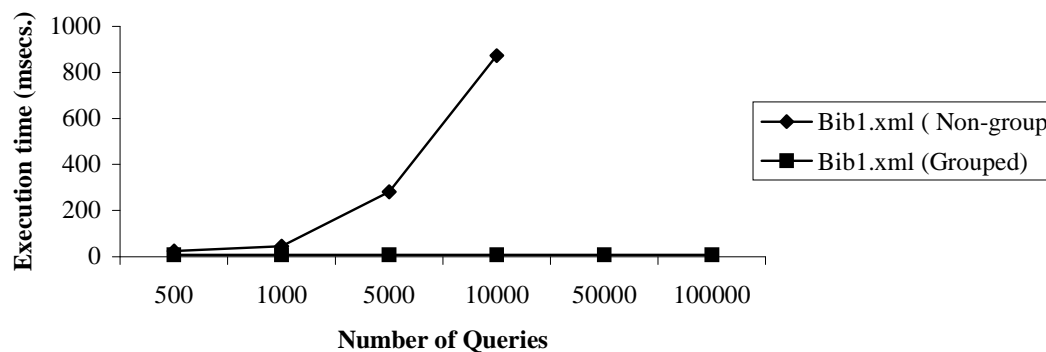

Figure 21. An experiment comparing the performance of grouped queries with non-grouped queries

5.2 Effect of Query Grouping for Different Document Types

In the second set of experiments we fix the number of queries to 100,000 and measure the execution time of the query groups for different size input documents. Figure 22 shows the results for this setting. These results indicate that performance is more sensitive to document size when the number of query groups increases. This result also confirms the importance of the query grouping.
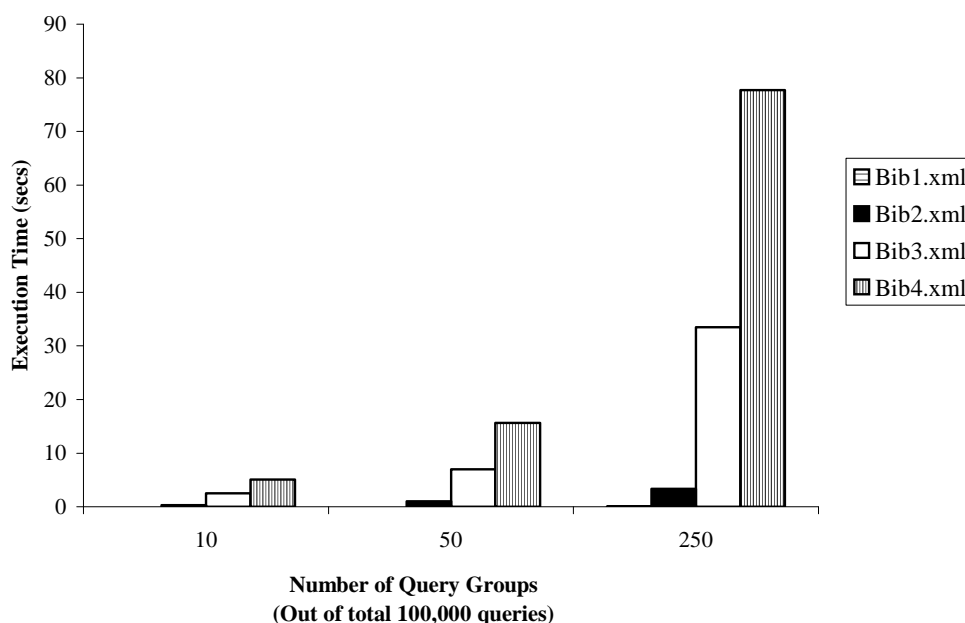

Figure 22. The execution times of queries for different number of query groups and document sizes

5.3 Effect of the Document Size

From the above experiments it is also clear that the size of the document effects the performance considerably. Figure 23 shows how the document size effects the performance. An important observation in these experiments is that large documents degrade the performance drastically when the number query group is high. Therefore, to increase the performance in this respect delta file option needs to be investigated.
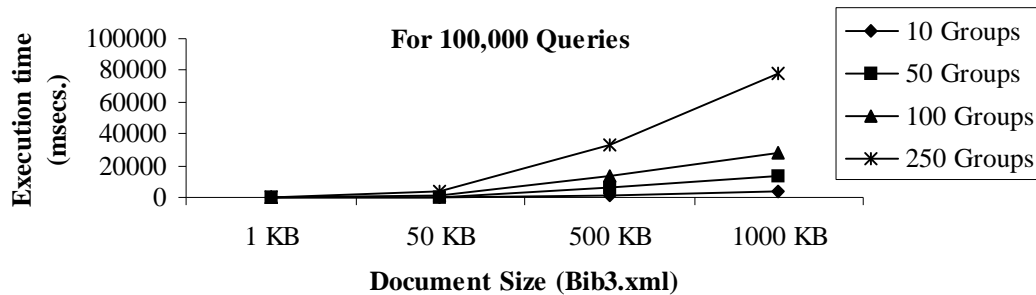

Figure 23. Results showing the effect of document size and query groups on performance

As final conclusion we can say that FSM approach described in this paper for executing XML-QL queries on XML documents residing in memory is a very promising approach to be used in mobile environments.

6. Delta Files

Using delta files rather than the original document is an issue to be investigated. A delta file demonstrates the changes between two versions of a document. A delta file format for XML proposed in [La Fontaine 01] is shown in Figure 24. This format indicates the changes through "modify", "add" or "delete" attributes for each changed element. Note that when there is a change in a child element, this change is propagated through its parents to indicate the change in the child element in the upper levels.

```
<!ELEMENT stock (symbol*)>
<!ELEMENT symbol (indexValue, changeRatio)>
<!ATTLIST symbol
        name CDATA #REQUIRED
        sector CDATA #REQUIRED>
<!ELEMENT indexValue (#PCDATA)>
<!ELEMENT changeRatio (#PCDATA)>
```

Figure 24. The DTD of the Stock.xml

```
<stock d:delta="modify">
        <symbol name="abc" d:delta="modify">
                <indexValue d:delta="modify">
                        <d:PCDATAmodify>  <d:PCDATAold>105</d:PCDATAold>
                                          <d:PCDATAnew>110</d:PCDATAnew>
                        </d:PCDATAmodify>
                </ indexValue >
                < changeRatio d:delta="modify">  <d:PCDATAmodify>
                                                  <d:PCDATAold>0.05</d:PCDATAold>
                                                  <d:PCDATAnew>0.1</d:PCDATAnew>
                        </d:PCDATAmodify>
                </ changeRatio >
        </ symbol >...  </stock>
```

Figure 25. An example delta file

The Query Execution Engine of the CQMC system requires minor modifications to handle the additional tags (delta tags) to process delta files. The algorithm handling the delta files is presented in the Appendix II. In the following we demonstrate the process through an example. Consider the query given in Figure 26 and the delta file given in Figure 25 corresponding to the DTD given in Figure 24.

```
WHERE
<symbol name ="abc">
        <indexValue>$a</indexValue>
        < changeRatio>$b</ changeRatio>
</symbol>
CONSTRUCT
<result>
        <abc>
        <indexValue>$a</indexValue>
        < changeRatio>$b</ changeRatio>
        </abc>
</result>
```

Figure 26. An example query

Note that since both the old and the new values of elements satisfy the query condition which is name= "abc", the result set obtained from the new data in delta file can be used to replace the corresponding elements in the original result set.

However it should be noted that not all queries can be handled through delta files. Consider the example query in Figure 27. Since the delta file shows only the changed portions of the XML file, and the "sector" attribute of the element "symbol" has not changed in the document, there is no way to run the query given in Figure 26 on the delta file although its result might have been effected.

```
WHERE
<symbol sector ="xyz">
</symbol> ELEMENT AS $a
CONSTRUCT
<result>$a </result>
```

Figure 27. An example query

There seems to be an obvious advantage to use a delta file rather than the original document when the size of the delta file is a lot smaller than the original file. However, as demonstrated above, whether a delta file will contribute to the result is not straightforward. Deciding on whether to use delta file involves a performance tradeoff depending on several metrics such as the size of the original file, the size of delta file, the number of queries involved as well as the complexity of the algorithm to check the applicability of the delta file to the queries at hand which are the issues that need to be further investigated.

7. Summary and Conclusions

Mobile communication is booming and access to Internet from mobile devices has become possible. Given this new technology, researchers and developers are in the process of figuring out what users really want to do anytime from anywhere and determining how to make this possible.

We anticipate that one of the common uses of mobile devices will be to deliver highly personalized information. There are in fact extensive efforts in this direction, however the level of personalization is limited to choosing from available content links, icon and services. We believe that a querying power is necessary for expressing highly personalized user profiles and for the system to be of use to millions of mobile users, it has to be scalable. Since the critical issue is the number of profiles compared to the number of documents, indexing queries rather than documents makes sense.

This paper describes such an architecture for mobile network operators for delivering highly personalized information from XML sources to mobile clients. The users are provided graphical user interfaces to define their profiles from their desktops. Simple profiles can also be defined from

mobile devices through WML. These profiles are converted into XML-QL queries. The queries can be change based or timer queries; that is, they need to be activated either when the related XML documents change or the time to execute the query expires. The queries are grouped and indexed such that each element in a query group corresponds to a state in the Finite State Machine (FSM) corresponding to that query. The system also contains an XML repository. When either there is a change in related XML documents or a timer based query (or a set of queries) expires, an event based XML parser is activated that starts sending the events to the Query Execution Engine component of the system that causes the related FSMs to change their states. The Query Execution Engine captures the intermediate results during state changes. It should be noted that all the queries that apply to a document are executed in parallel when a document is being parsed. The results produced are pushed to the related mobile clients. Experimental results presented in Section 5 proved that the query grouping and indexing mechanisms presented in this paper provide an excellent performance for different query and XML document workloads.

References

[AF 00] M. Altinel, M. J. Franklin, "Efficient Filtering of XML Documents for Selective Dissemination of Information", in Proc. of VLDB 2000, Cairo, September 2000.

[BW 01] S. Banu, J. Widom, "Continuous Queries over Data Streams", ACM Sigmod Record, Vol. 30, No.3, September 2001.

[CD 99] J. Clark, S. DeRose, "XML Path Language (XPath) Version 1.0", W3C Recommendation, http://www.w3.org/TR/xpath, November, 1999.

[CDTW 00] J. Chen, D. DeWitt, F. Tian, Y. Wang, "NiagaraCQ: A Scalable Continuous Query System for Internet Databases", ACM SIGMOD Intl. Conf. on Data Management, Texas, USA, June 2000.

[CHTML] Compact HTML for Small Information Appliances, *http://www.w3.org/TR/1998/Note compactHTML 19980209*, February 1998.

[FK 99] D. Florescu, D. Kossmann, "Storing and Querying XML Data using an RDBMS", IEEE Data Engineering Bulletin, Vol. 22, No.3, pp27-34, 1999.

[FKMX 00] D. Florescu, D. Kossmann, I. Manolescu, F. Xhuman, XML and Relational: How to live with both, in Proc. of VLDB 2000, Cairo, September 2000.

[GPRS 00] http://horizongprs.motorola.com/whatisgprs/whatis1.htm

[IDC 00] http://www.idc.com/eBusiness/press/ EBIZ082200pr.stm

[Kilic 01] O. Kilic, Profile Generation in Continuous Query Environments for Mobile Clients (CQMC), MS Thesis, Dept. of Computer Eng., Middle East Technical University, September 2001.

[La Fontaine 01] Robin La Fontaine**,** http://www.gca.org/papers/xmleurope2001/papers/pdf/s29-2.pdf, XML Europe 2001 Conference Proceedings, 21-25 May 2001.

[Meg 98] Megginson Technologies, "SAX 1.0: a free API for event-based XML parsing", http://www.megginson.com/SAX/index.html, May, 1998.

[MW 99] J. McHugh, J. Widom, "Query Optimization for Semistructured Data", in Proc. VLDB, 1999.

[NA 00] Nokia Artus Messaging Platform, http://www.nokia.com/networks/17/maxp.html, 2000.

[Ozen 01] T. Ozen,, Profile Processing in Continuous Query Environments for Mobile Clients (CQMC), MS Thesis, Dept. of Computer Eng., Middle East Technical University, September 2001.

[WAP 00] Wireless Application Protocol, http://www.wapforum.org, June, 2000.

[WML 00] Wireless Markup Language Specification, http://www1.wapforum.org/tech/documents/WAP-191-WML-20000219-a.pdf, February, 2000.

[XHTML] XHTML Basic, *http://www.w3.org/TR/2000/REC-xhtml-basic-20001219*, December, 2000.

[XML 98] Extensible Markup Language, http://www.w3.org/XML/, February, 1998.

[XML-QL 98] XML-QL: A Query Language for XML, http://www.w3.org/TR/1998/NOTE-xml-ql-19980819, August, 1998.

[XSL 00] http://www.w3.org/Style/XSL/

Appendix I: Algorithm Query Execution Process

No level check is performed for the root nodes as well nodes having the value –1 for relative position

Start Element Handler
1. Write start element tag into content of variables.
2. Find element in the *Query Index*
    2.1 For each node in the *Candidate List* (**CL**) of the element,
        2.1.1   Perform Attribute Filter check and Perform level check
            2.1.1.1   If the current node is a root node then
                2.1.1.1.1   If the node has a level info and current element level > node level, then copy the current node into the **CL.** Make this node the current node (Move cursor to the this node).
                2.1.1.1.2   Update level info of the current node
            2.1.1.2   If the node has character data, then set Process Flag to **TRUE**
            2.1.1.3   If the node has variable, then
                2.1.1.3.1   Set Process Flag to **TRUE**
                2.1.1.3.2   Create an empty content for the variable and start collecting content.
                2.1.1.3.3   If the type of the variable is **ELEMENT_AS,** then write start tag into content of activated variable.
            2.1.1.4   Copy the next nodes of the current node into their **CL**s. Update the level of next nodes.

End Element Handler
1. Find element in the Query Index
    1.1.   For each node (satisfying level check) in the Candidate List (**CL**) of the element
        1.1.1.   If the node has the variable and Process Flag is **TRUE**
            1.1.1.1. If the variable is an **ELEMENT_AS** varibale, then write the end element tag into content of activated variable.
            1.1.1.2. Set Process Flag to **FALSE**
            1.1.1.3. Stop collecting content
            1.1.1.4. If the next nodes of the node are in the **Final State**, or there is no next node
                then state flag of the node is set to **TRUE**
                else state flags of the next nodes are set to **FALSE**
        1.1.2   If the node is Normal node (simple tag <a>),
            1.1.2.1 If the next nodes of the node are in the **Final State**, or there is no next node
                then State Flag of the node is set to **TRUE**
                else State Flags of the next nodes are set to **FALSE**
        1.1.3 If the node is root node, then
            1.1.3.1 if the root node is on final state then collected contents in content list is written to output.
            1.1.3.2 Reinitialize the nodes of the query (reinitialize FSM)
            1.1.3.3 Clean the next nodes of the current node from **CL**
            1.1.3.4 If it is the copied root node, then delete all the next nodes and delete the node.
        1.1.4 Write end element tag into contents of activated variables

Element Data Handler
1. Write element data into content of activated variables.
2. Find the element in the *Query Index*
    2.1 For each node in the **CL**
        2.1.1   If it has the character data,
            2.1.1.1 If character data is matched then set state flag of the node and state flag of the query group to
                TRUE

End Document Handler
1. Finalize output generation.
3. Signals delivery component.

Appendix II: Algorithm to process delta files

```
for every element in delta file
        if the element is inserted
                if values of the element satisfies the query conditions
                        append result obtained from the element to the previous
                        result set
        if the element is deleted
                if values of the element satisfies the query conditions
                        remove result obtained from the element from the previous
                        result set
        if the element is modified
                if old values of the element satisfies the query conditions and
                   new values of the element dissatisfies the query conditions
                        remove result obtained from the element from the previous
                        result set
                if old values of the element dissatisfies the query conditions and
                   new values of the element satisfies the query conditions
                        append result obtained from the element to the previous
                        result set
                if old values of the element satisfies the query conditions and
                   new values of the element satisfies the query conditions
                        replace result obtained from the element with corresponding one in
                the previous result set
```