

**DLAlert - AN INFORMATION ALERT
SYSTEM FOR DIGITAL LIBRARIES**

by

GIANNIS ALEXAKIS

Submitted in partial fulfillment of the
requirements for the diploma of

Electronic and Computer Engineering

Technical University of Crete

June 2003

Guidance Committee :

Manolis Koubarakis	Associate Professor (supervisor)
Stavros Christodoulakis	Professor
Euripides Petrakis	Associate Professor

Abstract

As information available on the Internet is increasing from day to day, a user has to spend a lot of time searching, browsing and rejecting useless information in order to stay up to date, until he finds exactly what he is looking for. A new type of web applications called alerting services, assume the responsibility to collect all relevant data in a specific area and deliver them to each user regularly according to his fields of interest.

Alerting services could prove to be very helpful in the area of digital libraries, as the quantity of scientific publications doubles every 10-15 years and classic search applications become more and more ineffective to handle this information overload on their own. In this text we describe the design and implementation of DLAlert, an alerting system for digital libraries developed for the Library of the Technical University of Crete and ready to support many other sources including libraries, publishing houses and other alerting services. DLAlert is a web application that receives requests through a web page about the publications that interest every user, stores profiles about every user's fields of interest, collects information about new publications from the Technical University Library gateway, produces notifications for each user and sends an appropriate e-mail containing all relevant bibliographical data.

Acknowledgements

I would like to express my grateful thanks to the following people for their help, advice and information in developing this application.

My sincere gratitude to my supervisor Manolis Koubarakis
for his precious guidance and advice.

Stamatis Andranakis for his help during the implementation of the Z39.50 client.

Contents

Chapter 1	
Introduction	6
1.1 Alerting applications – description	6
1.2 Alerting applications – examples	7
1.3 DLAlert - Alert system for Digital Libraries.....	8
1.4 Organization of the missertation.....	9
Chapter 2	
Related Work	10
2.1 Alerting applications on the web.....	10
2.1.1 Elsevier Contents Direct	11
2.1.2 Kluwer Alert	12
2.1.3 Springer Alert.....	13
2.1.4 Hermes	14
2.2 DIAS (Distributed Information Alert System)	15
2.2.1 The WP (Word Pattern) data model	16
2.2.2 The AWP (Attribute based Word Pattern) data model.....	18
2.2.3 The AWPS data model	19
2.2.4 Some interesting problems	21
2.3 The Z39.50 protocol	21
2.3.1 Initialization facility.....	23
2.3.2 Search facility	24
2.3.3 Type-1 query syntax	26
2.3.4 Retrieval facility	28
2.3.5 Record format (UNIMARC).....	29
2.3.6 Z39.50 and interoperability	30
2.4 Oracle Text and filtering applications	31
2.4.1 The CTXRULE index	32
2.4.2 Creating the tables	35
2.4.3 Language of the stored queries.....	35
2.4.4 Indexing the stored queries	40
2.4.5 Filtering.....	41
2.5 Conclusions.....	43
Chapter 3	
System Overview	44
3.1 DLAlert architecture.....	44
3.2 Main Technologies used	46
3.3 Graphical User Interface overview	48
3.4 The language of the text queries	55
3.5 Conclusions.....	56

Chapter 4	
Database Schema.....	57
4.1 Requirements analysis	57
4.2 Relational schema.....	59
4.3 Key consistency and atomic transactions	60
4.4 Indexing of the stored queries	61
4.5 Conclusions.....	63
Chapter 5	
PL/SQL packages.....	64
5.1 Filtering module.....	64
5.1.1 The algorithm.....	64
5.2 Notifying module.....	68
5.2.1 The UTL_SMTP package.....	68
5.2.2 Collecting the matched publications for a single user	69
5.3 Performance.....	72
5.4 Conclusions.....	74
Chapter 6	
The Graphical User Interface.....	75
6.1 Middle application tier architecture.....	75
6.2 The Enterprise Java Bean.....	77
6.3 OC4J custom tag library.....	80
6.4 Preventing CTXRULE index errors	81
6.5 Parsing the text queries.....	86
6.6 Conclusions.....	88
Chapter 7	
The Observer.....	89
7.1 Information providers.....	89
7.2 Observer architecture.....	90
7.3 JZKit API	92
7.4 UNIMARC parser	93
7.5 SQLJ functionality	95
7.6 Performance.....	97
7.7 Important technical issues.....	97
7.8 Conclusions.....	98
Chapter 8	
Scheduling DAlert.....	99
8.1 Simple scenario.....	99
8.2 Supporting three types of desired notification frequencies.....	100
8.3 Conclusions.....	101
Chapter 9	
Concluding remarks	102
9.1 Future work on DAlert	102
9.2 Conclusion.....	108
Bibliography	109

The main difference with classic information retrieval applications is that the former evaluate every user's request only once and send the results immediately. In a selective *dissemination of information* or *alert system* the queries are stored and the user is notified every time there is a new event or data that might interest him.

1.2 Alerting applications – examples

Alerting services are becoming more and more helpful and selective dissemination of information techniques can be applied into many domains. Consider the following examples:

A news portal broadcasting e-mail messages to registered members adapted to every user's fields of interest containing the daily news.

An e-commerce company integrating information about merchandise from various providers and sending advertisements according to every customer's previous shopping.

A stock exchange company alerting stockholders upon certain events in the stock market (an increase or decrease of current rates and prices).

Every application like the ones above has its own characteristics:

The way the system collects information from various *sources*. Sources could be data retrieved from databases representing events on the real world. Every selective dissemination service monitors different events according to its specific area of interest.

The system should provide a standard way that users can request this information and define the conditions upon they want to be notified. The so-called *language of the profiles* is the language that the stored - queries are defined. Queries can be expressed directly in expressions that the system can recognize, by using a simple user-friendly graphic interface with buttons and drop – down lists or even implicitly by monitoring the user's previous actions (like the second example above).

1.3 DLAlert - Alert system for Digital Libraries

The number of scientific publications is estimated to double every 10-15 years. This means that the number of scientific papers, journals and books published before 1990 is close to the number published the last decade. It becomes harder for people to follow this evolution of technology without spending a lot of time daily on the Internet searching and browsing. Additionally knowledge of technology is provided often by independent miscellaneous organizations (universities, publishing houses, research departments in companies etc.). As new technological branches appear every day, there is need for tools that help people navigate through all this available information. Search engines dedicated to literature (scientific or not) and alerting applications in the same area are becoming very popular tools on today's Internet.

This dissertation presents the design and implementation of DLAlert, an alert system for digital libraries developed for the Library of the Technical University of Crete and ready to support many other sources including libraries of other organizations and alerting services.

Features distinguishing the aforementioned system include

The events that interest a potential user are new publications.

The sources are the bibliographical attributes of the new publications inserted in the digital libraries supported by DLAlert (up to now only the Library of the Technical University of Crete).

The language of profiles offers queries about words, phrases or concepts contained in the publications bibliographical attributes. Additionally, it provides *Boolean* and *proximity* operators for definition of relations between the previous terms.

The user is expected to describe his fields of interest by typing his queries in a *graphical user interface* on the web (<http://intelligence.tuc.gr/alert/login.html>).

Notifications are well-formed e-mail messages send to the user containing all relevant *bibliographical attributes* of the matched publication.

1.4 Organization of the dissertation

This dissertation is organized as follows. In Chapter 2 we present related work in the areas of information retrieval and selective dissemination of information. In Chapter 3 we reference modeling and design issues about DLAlert. Chapter 4 presents the internal structure and organization of the database schema used for storing the profiles and filtering the incoming data. In Chapters 5-7 we explain briefly the operation of every one of the system's modules (filtering, notifying, web Graphical User Interface). In Chapter 8 we discuss possible ways to collect data from sources and present the current implementation that uses the Z39.50 standard to retrieve information from various digital libraries. Chapter 9 presents the actions performed by the system. Finally in Chapter 10 we present our conclusions and propose directions for future work on DLAlert.

Chapter 2

Related Work

2.1.1 Elsevier Contents Direct

Elsevier Contents Direct (<http://contentsdirect.elsevier.com/>) is the free e-mail alerting service from Elsevier Science which delivers notifications to users about new publications. The sources of this system are a large number of publications daily from various areas and providers. The web interface is as friendly and simple as it could be

2.1.2 Kluwer Alert

Kluwer Alert (<http://www.kluweralert.nl/>) is a service which promises to keep researchers on top of the latest in scientific publishing. The system collects a large

2.1.3 Springer Alert

Springer Alert (<http://www.springer.de/alert/>) is a service provided by Springer Science and supports a wide variety of sources. The queries are subject categories which are organized hierarchically. The user can also select the frequency of notifications and the preferred language of publications. The e-mails are sent regularly and contain only the title/author of the matched book or journal and a hyper-link to a web-page where one can read detailed description of the publication, download the table of contents and purchase it. The Springer Alert is the only service that provides promotional brochures and software via surface mail. Catalogue search for books and electronic media is a feature available in the same interface.

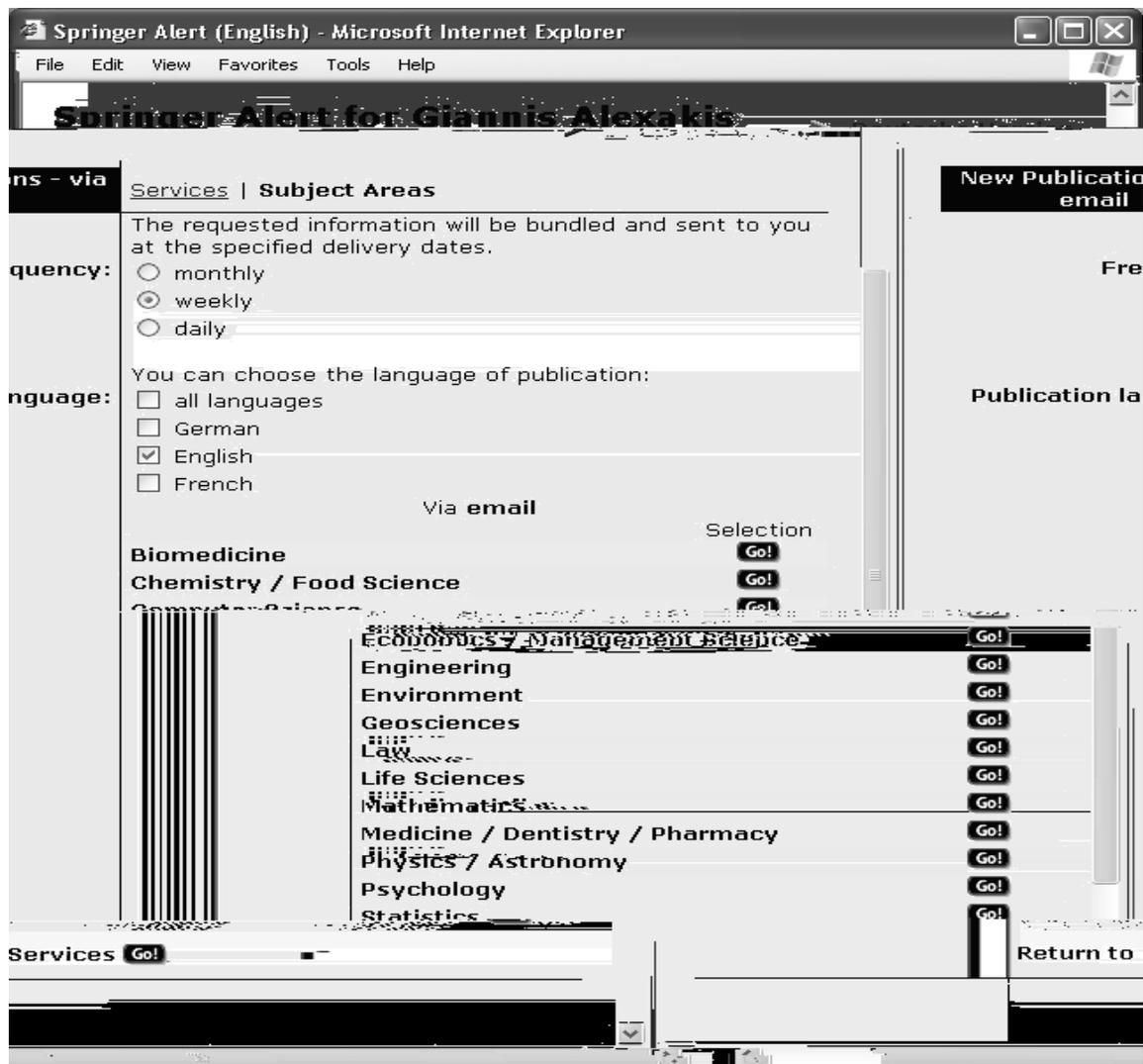


Figure 2.1-3 Springer Alert interface

2.1.4 Hermes

Hermes (<http://hermes.inf.fu-berlin.de/>) [4, 5] is an alerting service developed by the Institute of Computer Science of the University of Berlin. Hermes promises to integrate heterogeneous interfaces of different providers and support publishing houses or libraries that do not offer an alerting service themselves. Emphasis is on scholarly publications as journal articles, technical reports, and books.

It is the only service that provides specification of interest using an advanced mechanism. Profiles consist of one or more queries on bibliographical metadata and query terms as well as selection on specific journals. Queries on attributes can contain keywords or phrases related with Boolean operators (AND, OR, NOT). Queries to be stored are checked and those containing syntax errors are not inserted in the database of Hermes (an error message is produced). Single words standing alone or between Boolean operators are *stemmed* (for example the following expressions: library, libraries are equivalent).

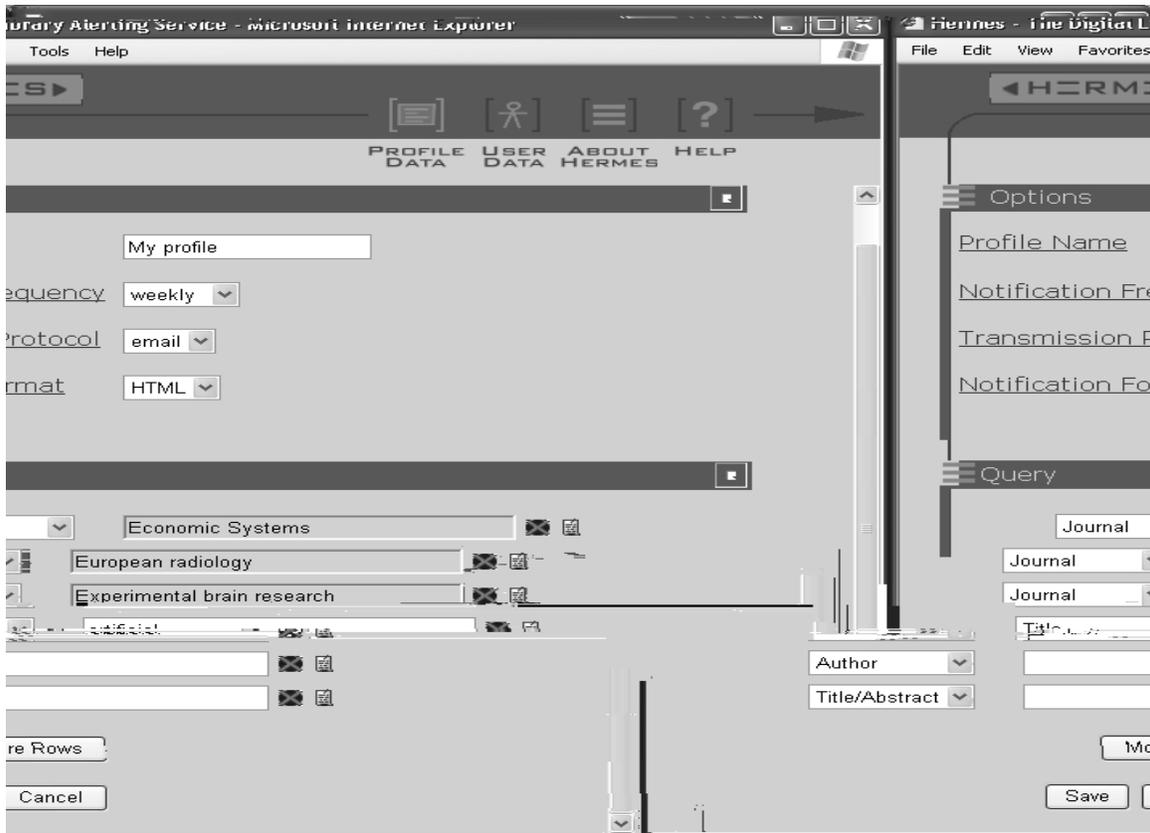


Figure 2.1-4 Hermes interface

The user can define *notification frequency* (day, week, month) and format (plain text, HTML, XML). The main disadvantage of this service is that the graphical interface is not as friendly and simple as the previous applications. The messages, which are usually not well formed, contain the main bibliographical attributes of the publications (title / author / abstract) and a hyper-link to detailed description. The system accepts *relevance feedback* on notifications which means that the user can evaluate the relevance of the delivered documents so that the *ranking results* are improved in later filtering. Generally speaking, Hermes is an application with advanced features but not as simple and friendly as the services presented earlier.

2.2 DIAS (*Distributed Information Alert System*)

DIAS [6, 7, 8] is a distributed alert system for digital libraries, currently under development in project DIET by the Intelligent Systems Laboratory of the Department of Electronic and Computer Engineering, Technical University of Crete. DIAS is currently implemented as a part of a system called P2P-DIET “A query and notification service based on mobile agents for rapid implementation of peer to peer applications”[52].

The advanced functionalities that DIAS offers and the basic ideas that this project proposes, motivated us during the implementation of DLAlert. Of course DLAlert is not a distributed system already, but the models and languages supported by both services are similar. In addition during our implementation stored queries from DIAS were used for validation of the filtering process of the DLAlert.

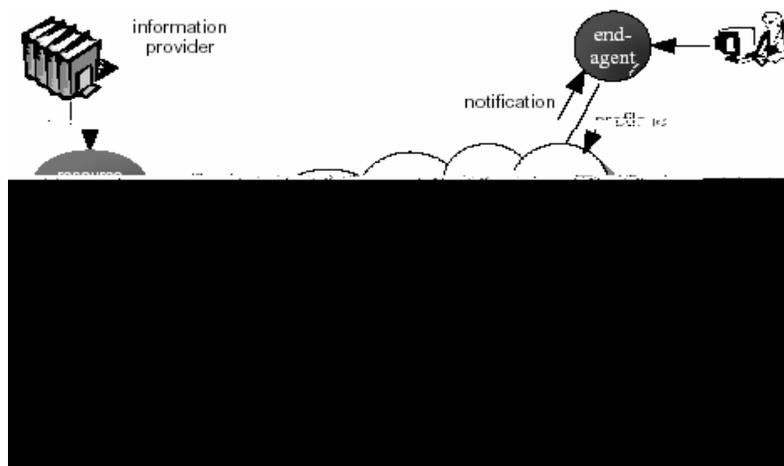


Figure 2.2-1 Architecture of DIAS

Before presenting the DIAS architecture in detail we have to mention the difference in terminology used. Notifications are considered not only the messages sent to the end-users but also all the messages exchanged between peers and can potentially contain information about new publications that must be disseminated through the network.

The architecture of DIAS is shown in Figure 2.2-1. *Resource agents* retrieve new publication's data from the information providers and produce streams of notifications containing this data. Users post profiles to some *middle-agent(s)* and receive notifications from the network, by using their *personal agents*. The notifications produced from the resource agents are propagated through the P2P network and arrive at interested subscribers (end-agents). Middle-agents forward the long-standing queries to other middle-agents in a way that matching of a profile with a notification takes place as close as possible to the origin of the incoming notification.

The models proposed by DIAS for notifications and profiles are presented below. We will not discuss the complete definition of these models and for more details on DIAS read [6, 7, 8]. In the following sections we present the schema for notifications every model proposes and the queries provided by its grammar. The definitions of the models are reproduced verbatim from the paper [6].

2.2.1 The WP (Word Pattern) data model

This model assumes that *textual information* (of notifications) is in the data of free text and can be queried by *word patterns*. A word w is considered a finite non-empty sequence of letters from a given alphabet. We also assume the existence of a (finite or infinite) set of words called the *vocabulary*. A *text value* s is a finite sequence of words from the assumed vocabulary. Thus $s(i)$ gives the i -th element of s and $|s|$ its number of words. The queries are word patterns generated according to the following grammars.

A *proximity-free word pattern* is an expression generated by the grammar

$$WP \quad w \mid WP \mid WP \quad WP \mid WP \quad WP \mid (WP)$$

A *proximity word pattern* is an expression $wp_1 \prec_{i_1} wp_2 \prec_{i_2} \dots \prec_{i_{n-1}} wp_n$ where wp_1, wp_2, \dots, wp_n are *positive proximity-free word patterns* (does not contain the

negation operator \neg). Where i_1, i_2, \dots, i_{n-1} are *intervals* (that represent order and distance between words) from the set I where

$$I = \{l, u : l, u \in \mathbb{N}, 0 \leq l < u\} \cup \{l, :l, :l, 0 \leq l\}$$

A word pattern is an expression generated by the grammar

$$WP \rightarrow PFWP \mid PWP \mid WP \mid WP \mid WP \mid (WP) \mid WP$$

P is a proximity free word pattern and PWP is a proximity word pattern.

2.2.2 The AWP (Attribute based Word Pattern) data model

The AWP data model defines that textual information on notifications is based on attributes or fields with finite-length strings as values. Strings will be understood as sequences of words (text values) as formalized by the model WP presented earlier. Attributes can be used to encode the bibliographical attributes of a publication (e.g., author, title, abstract of a paper and so on).

A notification schema N is a pair

Query examples for the AWP model

A [0,6]*TITLE* (John Smith)

the word `programming` in the attribute *TITLE*. This query matches the example notification (*TITLE* contains the word

AUTHOR = "John Brown" *ABSTRACT* \supset paper

matches notifications that the *AUTHOR* attribute is not "John Brown" and contain the

concept of

by assigning them a higher weight. Presenting the definition of this heuristic is out of the scope of this dissertation.

$sim (s_q, s_d)$ is a function that uses the weights of the words of two text values s_q, s_d to produce a number in the interval [0,1] that represents the concept of similarity between them

$$sim (s_q, s_d) = \frac{\sum_{i=1}^N q_i d_i}{\sqrt{\sum_{i=1}^N q_i^2 + \sum_{i=1}^N d_i^2}}$$

If the similarity value of two documents is close to 1 then these documents have similar semantic content.

The AWPS data model provides a new type of query that utilizes this function and issues requests on attributes that have similarity values over a certain threshold when compared with a given string. The syntax for this query is $A \sim_k s$ where A is an attribute, s is a text value and k is number in the interval [0,1] that gives a relevance threshold that candidate text values s should exceed in order to satisfy the predicate. A low similarity threshold k might result in many irrelevant documents satisfying a query, whereas a high similarity threshold would result in very few achieving satisfaction (or even no documents at all).

Query examples for the AWPS model

$$TITLE \sim_{0.6} \text{"Objemt Relational Databases"}$$

matches documents with *TITLE* relevant to "Objemt Relational Databases"

We should mention that we cannot give a notification that will always satisfy this predicate, because the similarity values of its attribute (*TITLE*) with the query string, depends on previously processed notifications. For example the notification bellow is most likely to satisfy the previous query

```
{(AUTHOR,Niemec      ),
(TITLE,"Objectriented Programming and Relational Databases  "),
(ABSTRACT,"...") }
```

Queries on similarity can be combined with Boolean operators and queries on word patterns.

$AUTHOR \sqsupset (John \prec_{[0,6]} Smith) \quad (TITLE \sim_{0.9} \text{"Artificial Intelligence"})$

matches notifications that contain the words John, Smith with 6 words between them or less in the attribute *AUTHOR* and *TITLE* relevant to “Artificial Intelligence”.

2.2.4 Some interesting problems

In the previous sections we presented in detail the language of the profiles used. DIAS also provides algorithms that efficiently solve the following problems

The satisfiability problem. As profiles and notifications propagate through the network a middle agent should be able to detect queries that could be satisfied by any notification at all.

The matching problem. Deciding whether an incoming notification matches a profile.

The filtering problem. Given a notification n an agent should be able to find all stored queries that match n .

The entailment problem. Deciding whether a profile is more or less “general” than another. An agent should detect profiles that request the same sets of notifications in order to minimize profile forwarding between peers.

.In DLAlert efficient filtering and matching are the necessary functionalities.

We have explained the language of the profiles provided by DIAS because queries

Z39.50 is an application layer network protocol (like HTTP, FTP, SMTP etc.) that uses the TCP/IP functionality and provides advanced information retrieval services, for organizations such

functionality. Instead of displaying the messages exchanged by the end-systems in raw data we present the output of the sample Z-client for convenience.

The functionality of Z39.50 is organized in “*facilities*”, which represent actions and consist of one or more *services*.

2.3.1 Initialization facility

The initialization facility is the action that establishes the connection (“*Z-association*”) between the client and the server. In the Init request, the client proposes values for *initialization parameters* (version of Z39.50, option flags, message sizes, other implementation information). The *option flags* indicate which other facilities are enabled during the Z-association. If the target requires authentication the origin should include a secret id / password in the request. In the Init response, the server responds with values for the initialization parameters; those values, which may differ from the client-proposed values, are in effect for the Z-association. If the server responds affirmatively (Result = ‘accept’), the Z-association is established. If the client then does not wish to accept the values in the server response, it may terminate the Z-association, via the Close service (and may subsequently attempt to initialize again). If the server responds negatively, the client may attempt to initialize again.

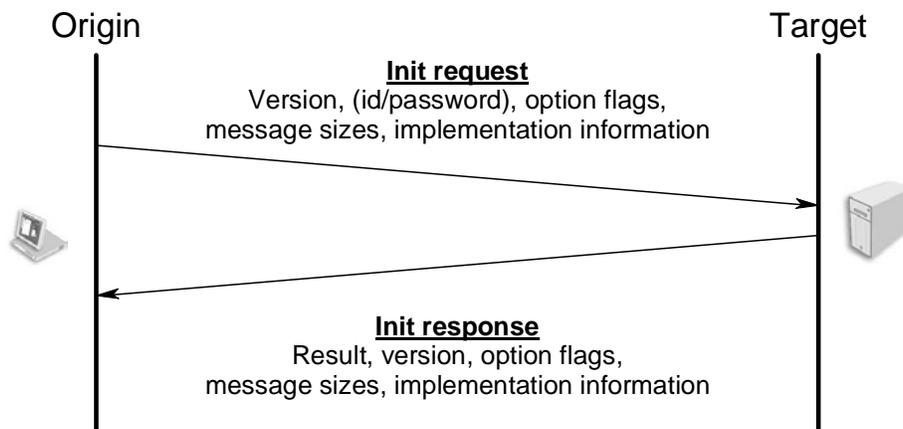


Figure 2.3-2 Initialization facility



Figure 2.3-3 JZKit sample client output

For example as we see in the previous picture by connecting to the Technical Universities Gateway (issue the command “open dias.library.tuc.gr”) we get the response that contains the implementation id : “1995” , the name: “Geac Advance Z39.50 Server” and version “2.0”. The target enabled services are Search, Present, Delete Result Set, Scan, Sort, Extended Services, Named Result Sets. In this dissertation we present only the Search and Present services because these are the only ones needed for the retrieval of new records for the purposes of DLAlert. More information on other Z39.50 services can be found in [1].

2.3.2 Search facility

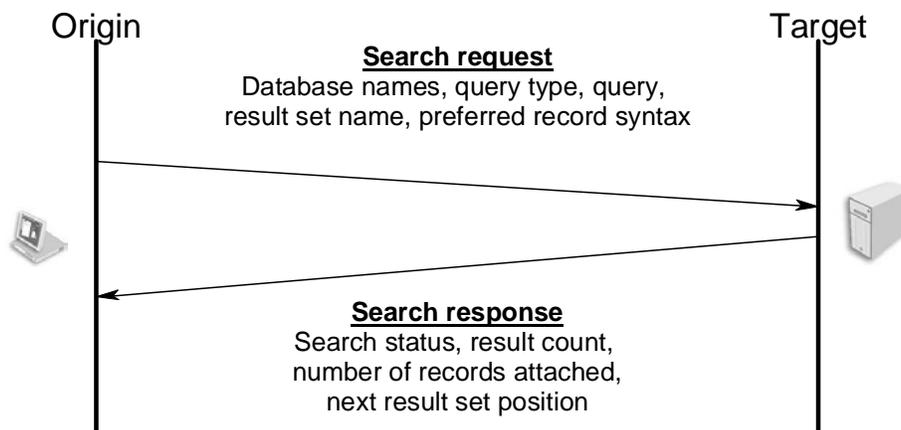


Figure 2.3-4 Search facility

The search facility is the action that sends a query on *abstract records* to the gateway. The origin sends the *database name*, *query-type*, *query*, the *result set name*

and *preferred record syntax*. The database name is sent because the gateway can be a front end to multiple databases and the query can refer to all or some of them. The query type used is *Type-1* (the default setting of the client) because it is supported by both the TUC gateway and the JZkit, provides the functionality we need for the purposes of DLAlert and is the most common query type used. We focus on this query type and give a detailed definition in the following section. The result set name is a string generated by the client so that the results of a search can be referenced. Preferred record syntax is UNIMARC [16, 17], the only one supported by the TUC gateway. The target returns *search status, result count, number of records attached and next result set position*. The search status indicates where the search completed successfully or not. The result count is the number of records that satisfy the query sent earlier. The response can contain attached records (usually in case of one or two results). The parameter next result set position takes on the value M+1, where M is the position of the result set item which identifies the database record corresponding to the last response record among those returned. Usually takes the value “1” in case the response does not contain attached records. The result records of a query are requested using the Present facility explained later.

```

C:\WINDOWS\System32\cmd.exe - java com.k_int.z3950.client.Z...
ZClient > base advance
dbnames:[advance]
ZClient > format unimarc
...
rch Response
erence ID : Search:0
rch Result : true
ult Count : 177
Records Returned : 0
t RS position : 1
nt >

```

Figure 2.3-5 JZKit sample client output

For example we connect to the digital library named “Advance” of the Technical University of Crete with the command “base advance” and define the preferred record format “format UNIMARC”. Then we send the query “@attrset bib-1 @attr 1=1035 smith” with the command “find”. This query requests bibliographical records that contain the word smith in any attribute. The response contains: the name of the result set “Search:0” (a string generated by the Jzkit), the status (“true”) that indicates

that the search completed successfully and the number of records satisfying the query “177”. The number of records returned is 0 so the next result set position is 1.

2.3.3 Type-1 query syntax

The Type-1 [18] query is also called RPN (*Reverse Polish Notation*) string because the operators must always be before the two related operands. An RPN string is generated according to the following grammar. Reserved words used might be slightly different among other Z39.50 API implementations but the grammar is a part of the protocol.

$$\begin{aligned} rpn - string & \quad @attrset \quad default - attrset \quad expr \\ & \quad default - attrset \quad bib-1 \end{aligned}$$

The *access points* to the abstract database are called attributes are categorized in *attribute sets*. The most common attribute set used in information retrieval from digital libraries is the *bib-1* [19] attribute set. The *bib-1* attribute set includes access points to attributes of bibliographic records. Other attribute sets could reference extended services tasks (*ext-1*), details of the target implementation (*exp-1*) or different organization of the access points to bibliographical records (GILS, CCL). A full listing of all registered attribute sets and generally all Z39.50 object identifiers can be found at [20]. Almost all Z39.50 implementations support the *bib-1* attribute set.

$$\begin{aligned} expr & \quad boolean \mid attr - plus - term \\ attr - plus - term & \quad attrdef \quad single - term \mid quoted - string \\ attrdef & \quad @attr \quad attrtype \quad attrval \\ boolean & \quad operator \quad expr \quad expr \\ operator & \quad @and \mid @or \mid @not \end{aligned}$$

single - term is considered a single word and *quoted - string* a set of words (enclosed in “ ”) that should be contained in the corresponding attribute of a record in order to satisfy the statement. *attrtype* for the *bib-1* attribute set is a value between 1 and 6 that describes the type of the attributes used. For *attrtype* 1 we can reference several

```
@attrset bib-1 @attr 1=32 2003
```

returns bibliographical records that contain the number 2003 in the date of acquisition field (records acquired in the year 2003).

```
@attrset bib-1 @or @or @attr 1=4 science @attr 1=4 algebra
@attr 1=4 mathematics
```

returns bibliographical records that contain at least one of the three words in the title.

2.3.4 Retrieval facility

The Retrieval facility is the action where the origin requests the results of a query from 001target. It consists of the Present and Segment services. In the Present service the client sends to the gateway the result set name referenced earlier in the Search facility, a number defining the starting point of the records, and the number of records to be returned. For example if a query returns 70 records and we want to retrieve the first twenty the starting point is 1 and the number of records is 20. T001target returns the records in a standardized format (XML, MARC, etc.), a number indicating the number of records returned and the status which indicates whether the Present service completed successfully.

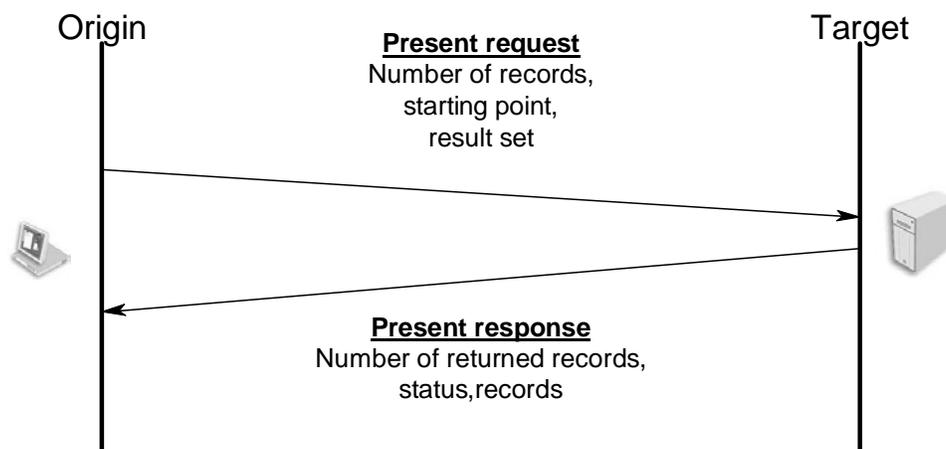


Figure 2.3-6 Present service

Sometimes the result set of a query may contain hundreds or thousands of records. The Present response could exceed an upper limit of bytes. Thus the server splits a Present

response that is larger than this limit into *segments*. In some Z39.50 implementations the origin could define the preferred sizes for message but the segmentation service in the target is responsible to decide the maximum segment size. In this case the number of records returned from the target is less than the number of records requested. Thus when constructing a client's Z39.50 request, we should be aware of the target's preferred record size. If the target's preferred record size is larger than the number of records requested, the target will return a response with a status of "records truncated" and the number of records returned will be less than the number of records requested.

ret(ined from a P rementr(eques(.))TJETq441.224 00 21-1.4 90 304.44cm/lm16 DoQfBT10.98 0 0 10.98 43

recordssZ Tw cotainf thewcorm smithe inany atear bue ("s

extended family of more than 20 MARC formats has grown up. Differences among various MARC formats meant that editing was required before records can be exchanged. One solution to the problem of incompatibility was to create an international MARC format (UNIMARC) [16, 17] which would accept records created in any MARC format. So in 1977 the International Federation of Library Associations and Institutes (IFLA) published UNIMARC: Universal MARC format, stating that "The primary purpose of UNIMARC is to facilitate the international exchange of data in machine-readable form between national bibliographic agencies".

The records retrieved from the Technical Universities Library are in the UNIMARC standard. The record structure is designed to control the representation of data by storing it in the form of strings of characters known as *fields*. The fields, which are identified by three-character numeric tags, are arranged in *functional blocks*. These blocks organise the data according to its function in a traditional catalogue record.

Tag num.	Description	Tag num.	Description
0XX	Identification block	5XX	Related title block
1XX	Coded information block	6XX	Subject analysis block
2XX	Descriptive information block	7XX	Intellectual responsibility block
3XX	Notes block	8XX	International use block
4XX	Linking entry block	9XX	Reserved for local use

T a b l e 2 . 3 - 2 U N I M A R C f u n c t i o n a l b l o c k s

Within each field, data is coded into one or more subfields, e.g. 700 \$a ... \$b ..., etc., according to the kind of the information. The effect of the subfield coding is to refine further the definition of the data for computer processing. The subfield identifiers consist of a special character, represented by a \$ in the examples, and a lower case alphabetic character or a number 0-9. For example the field starting with the tag 210i contains publication related data and the subfield \$d contains publication date. We do not present the whole definition of UNIMARC format because it defines thousands of tags and subfields [17]. The main UNIMARC tags used by the TUC library and the corresponding Z39.50 attributes are shown in Section 2.3.3.

2 . 3 . 6 Z 3 9 . 5 0 a n d i n t e r o p e r a b i l i t y

Most digital libraries round the world nowadays have a Z39.50 Gateway as a front-end. The organizations, universities, museums or publication houses that support this protocol are uncountable. Accessing all those digital libraries using a common way,

which is independent to the specific implementation of each database, is the main advantage of Z39.50 functionality. We indicatively report some digital libraries in Greece that support this standard and we have successfully retrieved records using the client we constructed.

Organization	Gateway host : port
University of Thessaly	library.lib.uth.gr : 210
University of Patras	pherusa.lis.upatras.gr : 210
University of Cyprus	194.42.4.129 : 210
University of Aegean	library.lib.aegean.gr : 210
Technical Chamber of Greece (TEE)	artemis.tee.gr : 21210
Panteion University	library.panteion.gr : 210
Ionian University	zante.ionio.gr : 210
Hellenic American Education Foundation	194.30.242.11 : 210

Table 2.3-3 Some Z39.50 Gateways in Greece

2.4 Oracle Text and filtering applications

Oracle Text [23-27] is a tool used in the Oracle RDBMS [21, 22] that enables us build *text retrieval* and *filtering applications*. Retrieval applications enable users to find documents that contain one or more search terms defined in a query. Text is a collection a documents in plain text, HTML, or XML. A filtering application stores queries in the database and finds those which match a certain document. DAlert and generally alerting services are considered filtering applications. The grammars of queries used in text retrieval and in filtering are similar and search terms could be simple words, phrases or *themes*. Themes define concepts inside a document. In the Figure 2.4-1 we present an overview of the architecture of a filtering application.

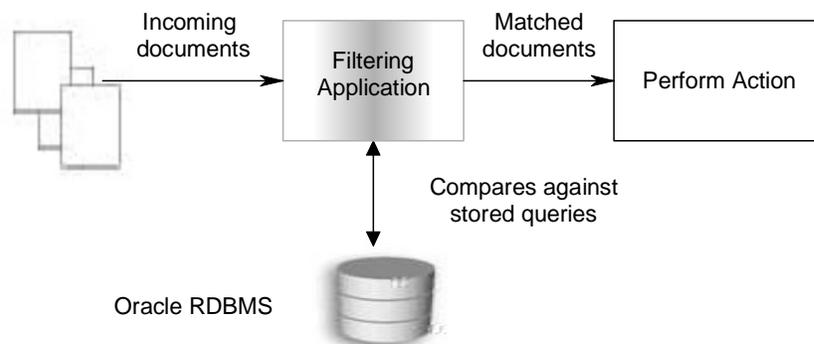


Figure 2.4-1 Filtering application

2.4.1 The CTXRULE index

The filtering functionality in Oracle database was first introduced in version 9.0.1 (June 2001) with the CTXRULE index type. In filtering applications queries are stored in a column of a table and a CTXRULE index should be constructed. This index is a structure that holds information about the stored queries. When Oracle finds maty4ap99 341. g 0 1.639

Datastore is the object which retrieves data from the table with the queries and creates a stream of query strings. The Lexer breaks the text into tokens according to our language. These tokens are usually words or query operators. The parser gets the queries from the Lexer, creates a parse tree and sends this back to the Lexer. The Lexer normalizes the tokens (turns into upper case, omits very frequent words like 'a', 'is' etc), breaks the parse tree into rules and sends these to the engine. The engine builds up an inverted index of rules, and stores it in the index tables.

For example we present the index content on some simple Boolean queries on keywords. The index is a structure that represents the parse tree generated by the query parser, containing among others the columns `TOKEN_TEXT` and `TOKEN_EXTRA`.

`QUERY_STRING` : the query string as it is stored in the base table
`TOKEN_TEXT` : the first token to be found for the query to be matched
`TOKEN_EXTRA` : the other tokens to be found for the query to be matched

Query 1 is a single word query. A document is a full match if it contains the word "oracle". In this case, matching `TOKEN_TEXT` alone is sufficient, so `TOKEN_EXTRA` is `NULL`. Notice the normalization of the token to upper case:

<code>QUERY_STRING</code>	<code>TOKEN_TEXT</code>	<code>TOKEN_EXTRA</code>
-----	-----	-----
oracle	ORACLE	(null)

Query 2 is an OR statement. A document is a full match if it contains the word "larry" or the word "ellison". This can be reduced to two single-word queries, each of which has `TOKEN_EXTRA NULL`:

<code>QUERY_STRING</code>	<code>TOKEN_TEXT</code>	<code>TOKEN_EXTRA</code>
-----	-----	-----
larry or ellison	LARRY	(null)
	ELLISON	(null)

Query 3 is an AND statement. A document must have both words "dbms" and "oracle" to be a full match. The engine will choose one of these as the filing term, and place the other the `TOKEN_EXTRA` criteria:

2.4.2 Creating the tables

For example let us define the following simple schema that consists of two tables: The table `Documents` (Table 2.4-1) with two columns: `article_id` primary key of the table, and `article_text` containing unstructured plain text of the incoming article. The table `Profiles` (Table 2.4-2) that holds the stored profiles of users and consists of two columns `query_id` primary key of the table and `query` the query itself. `article_id`, `query_id` are integers and `article_text`, `query` are strings.

<code>article_id</code>	<code>article_text</code>
12	Metals mining is the industrial sector responsible for the largest amount of toxic releases in the United States, according to a highly...
34	Papers in the robotics literature often concern specific technical aspects of robot research and development. At the same time, several robot competitions have emphasized ...
97	The Eighth Annual Mobile Robot Competition and Exhibition was held as part of the Sixteenth National Conference on Artificial Intelligence in Orlando...
80	The United States National Security Agency, with help from Network Associates of Santa Clara, Calif., has made a security-enhanced version of Linux available for download...

Table 2.4-1 Table Documents

<code>query_id</code>	<code>query</code>
...	...
311	toxic releases
312	US or Europe
313	Artificial AND Intelligence
314	near((personal, computers) , 4)
315	\$library
316	about(politics)
...

Table 2.4-2 Table Profiles

2.4.3 Language of the stored queries

Let us introduce the grammar which generates the queries for plain text documents. All operators are case-insensitive (*AND* is equivalent to *and*). All expressions that contain multiple tokens which are not connected with operators are considered terms (exact phrases). The available Boolean queries are shown in Table 2.4-2.

Function	Syntax	Description	Examples
	<i>term1</i>	matches documents that contain <i>term1</i>	toxic releases intelligence
conjunction	<i>term1 and term2</i>	matches documents that contain both terms	Artificial and Intelligence
	<i>term1 & term2</i>		modile & phone
disjunction	<i>term1 or term2</i>	matches documents that contain <i>term1</i> or <i>term2</i>	US or Europe
	<i>term1 term2</i>		Linux Windows
negation	<i>term1 not term2</i>	matches documents that contain <i>term1</i> but not <i>term2</i>	paper not journal
	<i>term1 ~term2</i>		software ~hardware

Table 2.4-2 Boolean queries syntax

Along with the standard Boolean queries, the CTXRULE index type grammar provides proximity, stemming and theme functionality.

➤ **Proximity operator NEAR (;)**

The operator *NEAR* matches documents based on the proximity of two or more query terms. The syntax of proximity queries is

$$n \text{ ar } ((t \text{ rm1}, t \text{ rm2}, \dots, t \text{ rmn}), \text{ max_span }, \text{ order })$$

term 1-n: the terms in the query separated by commas. The query terms can be single words or phrases.

max_span (optional – default 100): the maximum size of a clump where clump is the smallest group of words where all query terms occur. All clumps begin and end with a query term. *max_span* cannot be greater than 100.

order (optional – default FALSE): indicates whether terms are to be found in the same order as in the query.

Alternatively proximity can be defined according to the syntax

$$t \text{ rm1} n \text{ ar } t \text{ rm2}$$

$$t \text{ rm1} ; t \text{ rm2}$$

These queries are equivalent to the expression: $n \text{ ar } ((t \text{ rm1}, t \text{ rm2}), 100, \text{FALSE})$

Example	Description

Table 2.4-3 Examples of Proximity queries

➤ **Stem operator (\$)**

The stemming functionality enables us request terms that have the same linguistic root as the query term. The stem operator (\$) expands a query to include all terms having the same stem or root word as the specified term.

Example	Description
\$scream	matches documents that contain at least one of the terms "scream", "screamed", "screaming"
\$library	matches documents that contain at least one of the terms "library", "libraries", "librarian"
\$sing	matches documents that contain at least one of the terms "sing", "sang", "sang"

Table 2.4-4 Examples of Stemming queries

The definition of the algorithm used for stemming is not available in the manuals provided by Oracle [23-25, 27]. The Oracle Text stemmer supports the following languages: English, French, Spanish, Italian, German, and Dutch.

➤ **Theme indexing**

Oracle supplies a database of themes in English or French. Themes are tokens organized hierarchically and connected to each other with relations that describe their semantic content. Oracle Text supports the typical relations used by thesauri and is compliant with the ISO-2788 [28] and ANSI Z39.19 (1993) [29] standards. The relation definitions are reproduced from the ANSI Z39.19 specification [29].

These relations are

A Synonym (*SYN*) relation defines *equivalence* between two terms and connects terms with very close meaning like the ones bellow.

scary *SYN* fear, hiddenly *SYN* secrecy, drive-up *SYN* automobiles, lounge *SYN* rest.

A Broader Term (*BT*) relation defines a *superordinate semantic class* that the first concept belongs to. For example

lordships *BT* royalty and aristocracy, American Revolution *BT* military wars,
backward motion *BT* withdrawal, Bible *BT* sacred texts and objects.

A Narrower Term (*NT*) relation defines a *subordinate semantic class* that the first concept includes. For example.

Roman Catholicism *NT* papism, computers *NT* laptops, behaviour *NT* sympathy,
organized crime *NT* gangsters, cosmology *NT* astronomy

The *NT* and *BT* relations are symmetric. If and only if $X \text{ } BT \text{ } Y$, then $Y \text{ } NT \text{ } X$.

A Related Term (*RT*) relation implies *semantic overlap* (there is an element of meaning common to both terms). For example.

8r8r RTR

➤ **Operator ABOUT**

Another operator that uses the supplied thesaurus to expand theme queries is the *ABOUT(phrase)* statement. The *phrase* parameter can be a single word or a phrase, or a string of words in free text format. If the *phrase* parameter does match exactly a stored concept Oracle normalizes it and finds the stored concepts closer to the original string. For example before expansion “*politic*” is normalized to “*politics*” and “*national*” to “*nations*”. If normalization fails to find a concept describing *phrase* parameter the query is satisfied only in exact phrase match. Otherwise Oracle Text expands the queries using synonyms and narrower terms of the concepts inside the parentheses. The terms of the expansion could be connected to the original term directly or via another interjected term (expansion level > 1). The definition of the algorithm used for normalization and query expansion is not available in the manuals provided by Oracle [23-25, 27] and should be rather complex. We provide an example of the expansion of the word “*politics*” instead.

Expansion terms are separated by commas. The word “and” in expansion terms is not considered an operator.

Relation to term "politics"	Expansions of the query ABOUT(politics)
narrower terms level 1	civil rights, elections and campaigns, political parties, political scandals, political sciences, politicians, politicians and activists, revolution and subversion, world politics
narrower terms level 2	civil liberties, elections, human rights, insurgents, insurrectionary, insurrections, partisan politics, revolutionaries, revolutionists, terrorism
narrower terms level 3	terrorist activities, terrorist incidents, terrorists
narrower terms level 1 of "political advocacy"	animal rights, consumer advocacy
narrower terms level 2 of "political advocacy"	animal rights activists, animal rights movement, animal-rights activists, consumer activists, consumer advocates, consumer rights
both "politics" and "policymakers" narrower terms of "government"	policymakers

Table 2.4-6 Expansions of term "politics"

Important Notes:

- i. An *ABOUT* statement cannot contain the proximity operator *NEAR*. For example the query: “ *NEAR*((personal, computers) , 4) *AND* *ABOUT*(software) ” is not valid and cannot be parsed.
- ii. The *phrase* parameter in an *ABOUT* statement should be in lower case.
- iii. Inside thesaural statements (*SYN*, *BT*, *NT*, *RT* and *ABOUT*) any reserved words like (*AND*, *OR*, *NOT*, *NEAR*) are not considered operators but simple tokens.

➤ **Operator precedence**

Within query expressions with two operands, the operators have the following order of evaluation from highest precedence to lowest: *NEAR*, *NOT*, *AND*, *OR*. For example:

Query Expression	Order of Evaluation
w1 OR w2 AND w3	w1 OR (w2 AND w3)
w1 AND w2 OR w3	(w1 AND w2) OR w3
w1 NOT w2 AND w3	(w1 NOT w2) AND w3
w1 OR w2 NEAR w3	w1 OR (w2 NEAR w3)
w1 NOT w2 NEAR w3	w1 NOT (w2 NEAR w3)

Table 2.4-7 Operator precedence examples

Grouping characters can be used to control operator precedence. Grouping characters are parenthesis () and brackets [].

2.4.4 Indexing the stored queries

Let us consider the relational simple schema defined in Section 2.4.2. Before filtering the documents we must first index the queries in order to be able to collect matching profiles. This is done with the following command:

```
CREATE INDEX profile_index on profiles(query)
      INDEXTYPE IS ctxsys.ctxrule;
```

_r47-1.7268 st m120.2994 0 Tr47-1.7268 st m120.2:cng c7-06 1 (1 1 Tfn68 T.34)

Error Description	Query
missing parenthesis	(software design
<i>term2</i> missing	international and
<i>term1</i> missing	or mathematics
<i>about(...)</i> and <i>near(...)</i> inthesamefield	about(management) and near((financial, planning),4)
operator <i>between</i> RDBMS and near(...) missing RDBMS	near((data, warehousing),6)
<i>about</i> is an operator	journals about science

Table 2.4-8 Syntax error examples

Syntax errors cause index errors (upon index creation or synchronization) and then filtering may not produce the expected results. Also indexing of null fields produce errors. These errors can presented using a simple *SELECT* statement on the data dictionary view *CTX_USER_INDEX_ERRORS*. This view contains four columns which are name of the index, time of error, row id of error query and error message. This view is very helpful for program debugging and database administration but not automatic error detection on queries. Also detecting an error after the transaction is committed, is not the optimal way to solve this problem. An application that lets users define their profiles should include a mechanism that validates queries before insert/update transactions (Sections 6.4-6.5). The main disadvantage of Oracle Text is that it does not provide such functionality for the CTXRULE index.

2.4.5 Filtering

In order to filter the documents we use the SQL operator *MATCHES*. We use this operator to find all rows in a query table that match a given document. The document must be a plain text, HTML, or XML document. This operator requires a CTXRULE index on our set of queries.

The syntax for this operatentis

MATCHES returns 1 in case of matching and 0 for no match. This number cannot be assigned to variable because *MATCHES* does not support functional invocation. This operator can only be used in *SELECT* statement according to this syntax using the greater than zero condition ">0".

```
select column1,column2,.. columnN from table
where MATCHES( column, document VARCHAR2 or CLOB )>0
```

To find all matching profiles of the schema 2.1.2 for a given document *single_doc* we use the following procedure *find_matching_profiles()*

```
single_doc
```

To find matching profiles for all documents in the table, we use the procedure `find_all` which uses a `for...loop` that calls the previous procedure on all documents.

```
create procedure find_all as
begin
for current_document in
  ( select * from documents )
  loop
    dbms_output.put_line('Article:' || current_document.article_id)
    find_matching_profiles(current_document);
  end loop;
end;
```

Instead of displaying the results on the screen we could have declared other actions be performed, on every matching profile. The most common case in a complete filtering application is to insert the primary keys of the results into another database table for further processing by other database modules. Anyway, the important feature of the *MATCHES* statement is the ability to find all matching profiles for a certain document instead of periodically running the stored queries on the documents.

2.5 Conclusions

In this chapter we presented the previous developments on this topic that inspired and helped us implement DLAlert. We evaluated popular Alerting Services on the web in the area of Digital Libraries. Then we discussed DIAS, a distributed alert system for digital libraries, and explained in detail its language of profiles and its functionality. We presented the main facilities of the Z39.50 standard, used for publication record retrieval from digital libraries. Finally, we introduced filtering application construction with Oracle Text. In the rest of the dissertation we focus on the design and implementation of DLAlert starting with an overview of the system.

Chapter 3

System Overview

In this the following Chapters we introduce the main design issues that concerned us during the implementation of DLAlert. First we present an overview of its modules and all related Oracle database features used. Then we give a brief manual for the interface of DLAlert and the language supported.

3.1 DLAlert architecture

DLAlert, as a complete alerting service, consists of several components that cooperate with each other, in order to achieve the desired function. The main parts of this system are (shown in Figure 3.1-1):

- The Oracle Database is where the profiles and user data are stored. New publication records are also inserted inside the database and stored temporarily until the notification messages (e-mail) are produced and send to each user. The RDBMS is the core functionality of DLAlert.
- The Observer is the component responsible for collecting new publication records from a Z39.50 gateway (the TUC digital library), and inserting them into the Oracle database.
- The Filtering module classifies incoming documents according to the users' stored queries and finds matching profiles and related publication records.
- The Notifying module collects all matched profiles and sends a single message for each user, containing the bibliographical attributes of the relevant publication.

The Observer, the Filtering and Notifying modules communicate with each other by writing/reading from common records of the database. These modules are scheduled to perform actions sequentially (see Chapter 8) so that the results from a previous component are processed by the following one.

- The Application Server provides the necessary software infrastructure for developing and deploying the middle tier of DLAlert. in addition to providing the necessary forms for the transactions on the web (inserts/delete/updates of profiles, user credentials), the Alerting Service must check the queries to be
-

stored for syntax errors and maintain user session state (every user has access restricted to the profiles of his account).

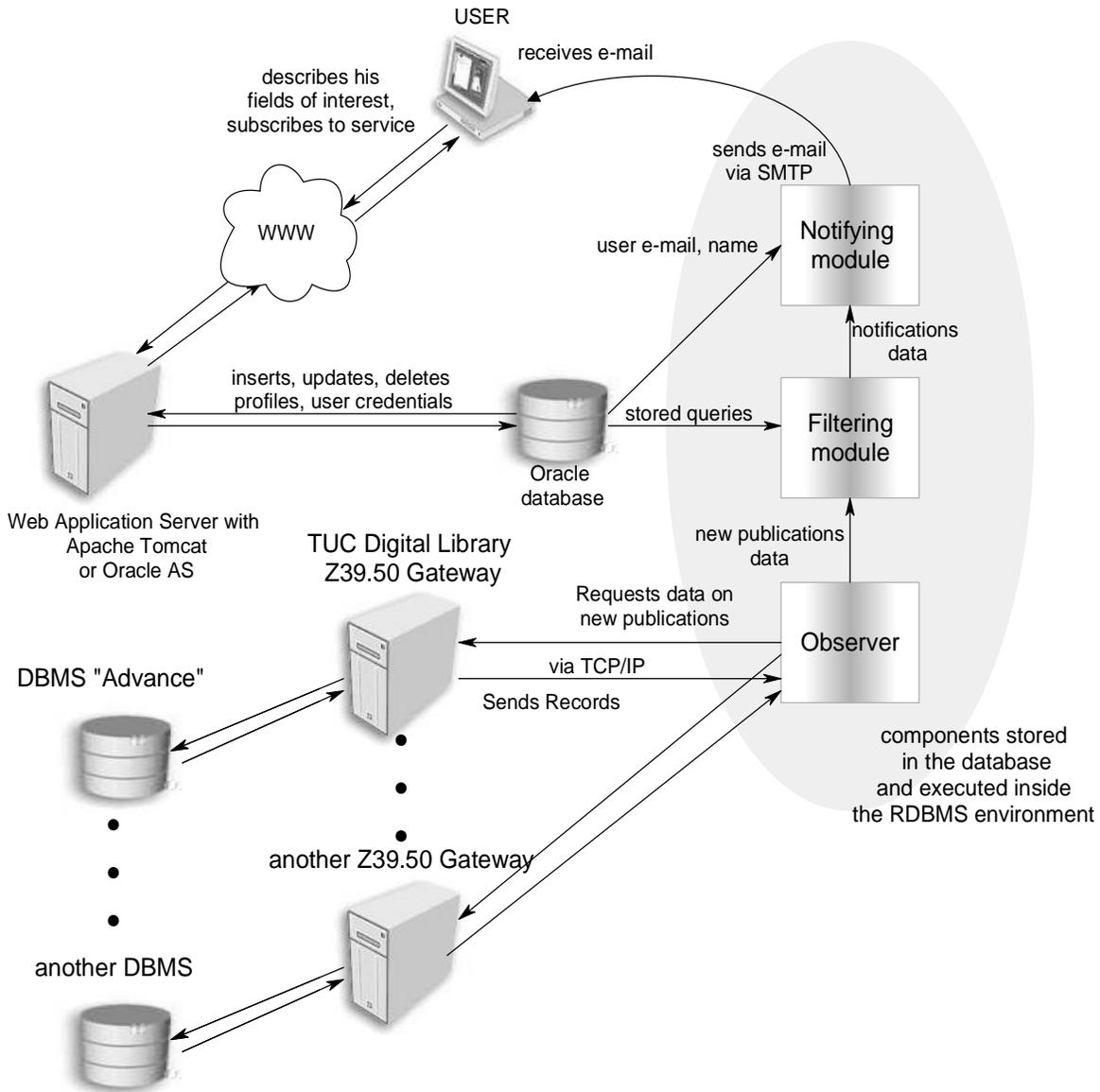


Figure 3.1-1 DLAlert architecture

The other parts of this schema are:

- The Z39.50 Gateways and DBMS of Digital Libraries as described in Section 2.3.
-

3.2 Main Technologies used

The Oracle database is the core of DLAlert and most of the applications programmatic code is stored, compiled and executed inside the database. Bellow we explain the advantages of using Oracle RDBMS as the essential component of our system.

- Nowadays most systems that require a robust and reliable centralized mechanism to store and retrieve large amount of data utilize a database. One of the main requirements of DLAlert is to be able to handle hundreds of thousands or millions of user profiles and hundreds of new publication records every day in a way that ensures the validity of the information stored. The necessity of using a database that can achieve these standards is indisputable. Any reliable RDBMS system provides functionalities that ensure data consistency and integrity, transaction concurrency and scalability for handling large amount of information. Using a software infrastructure like an RDBMS, the developer considers the aforementioned issues solved, and he is mainly concentrated on the particular requirements of his application.
 - *Oracle Text* (as shown in Section 2.4) is a specialized tool provided by the Oracle RDBMS which is able to provide document filtering techniques and profile matching functionalities, necessary for DLAlert. Without these mechanisms we should have constructed the filtering module of our system virtually from scratch. Oracle Text is the key feature that persuaded us to choose Oracle RDBMS among other equally reliable databases.
 - *PL/SQL* [30-32] is Oracle's procedural extension to industry-standard SQL. Its primary strength is in providing a server-side, robust procedural language. PL/SQL code is stored and executed inside the Oracle RDBMS environment and is responsible for the application's actions that involve data processing. The developer is able to construct procedures, functions and triggers using this language. Logically related stored procedures, functions and object types can be grouped into packages. The filtering and notifying module are PL/SQL packages.
 - Oracle RDBMS provides *PL/SQL Packages* [33] useful for application developers. In DLAlert two of them proved to be very helpful.
-

- *UTL_SMTP* is a package that provides functionality for sending e-mail messages over SMTP from the database. We utilized UTL_SMTP for sending the notifications on new publications to users.
 - *DBMS_JOB* can be used to schedule the execution of Oracle packaged procedures at a specific time and on a recurring basis. The collection of new publication records, document filtering and notification of users are operations of DLAlert that must be scheduled to run at certain intervals of time (for example once a day). Also we have to ensure these actions are performed one after the other so that the results a previous component are processed by the following one. The main advantages of using DBMS_JOB instead of any external scheduling application are security reasons (none can access the database without permission). Another important functionality is that DBMS_JOB can be programmed to detect unsuccessful completion of a module, and re-schedule it ensuring the proper queue of actions.
 - Java [34] is the most popular language used by application developers nowadays. One of the main advantages of Oracle RDBMS [35-39] is the ability to integrate Java classes inside the database and deploy them on a supplied Enterprise level Java 2 platform (Oracle Java Server). The Java code which can be loaded as either source or compiled (bytecode), is executed inside the database using the internal Oracle Java Virtual Machine. The accesses to the database use the server-side internal JDBC driver [35, 38], which means that the application can handle faster large amount of data than any external process. The static methods of any Java class can be declared stored procedures and can be even called from PL/SQL code. Although Java is not as fast as PL/SQL in case of intensive data access, is preferred if the application under development requires a complex computation mechanism or functionality not available in PL/SQL. DLAlert regularly
-

The other important component of DLAlert is the Application Server. The Application Server provides a J2EE (Java 2 Enterprise Edition) platform for developing and deploying web applications [40-42]. *Multi-tiered applications* (shown in Figure 3.2-1) dominate today's Internet. The *client tier* contains logic related to *presentation* and requests for services. The application server contains business logic that reads and writes data. In our case the Application Server provides the necessary *dynamic web pages* and *business logic* for data transactions to the Client (profiles, user credentials), restricts every users access to his account, checks queries for syntax errors and produces the appropriate error messages in case of invalid profiles. The internal architecture of the middle tier is explained in detail at Section 6.2.



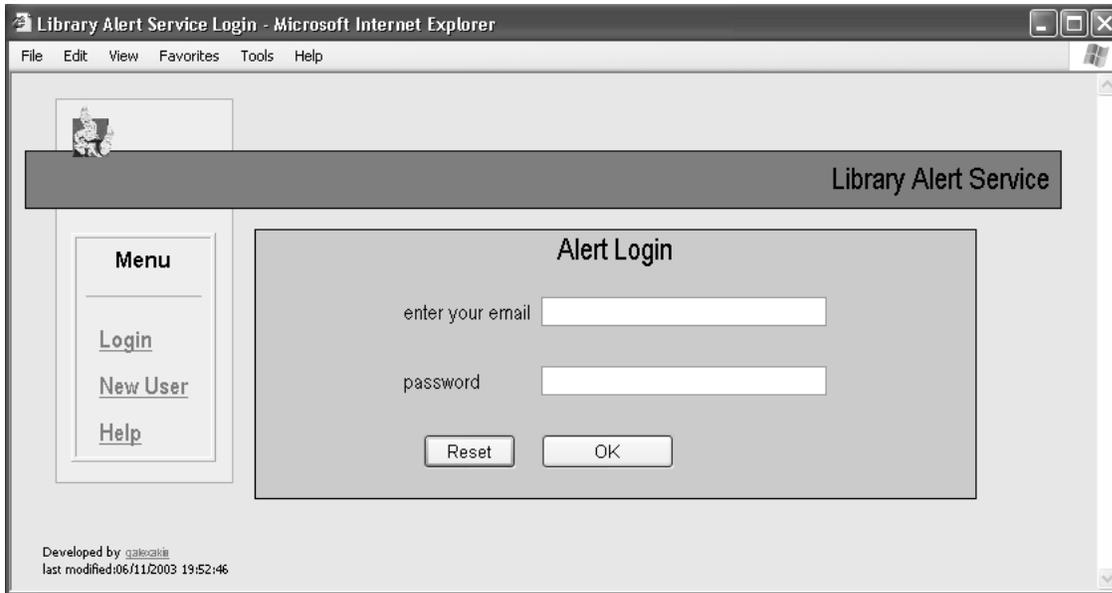


Figure 3.3-1 DLAlert: Login screen

A registered user would type his e-mail/password and login into his account. An unregistered user will click '*Help*' for more information about DLAlert or will click '*New User*' and enter his credentials.

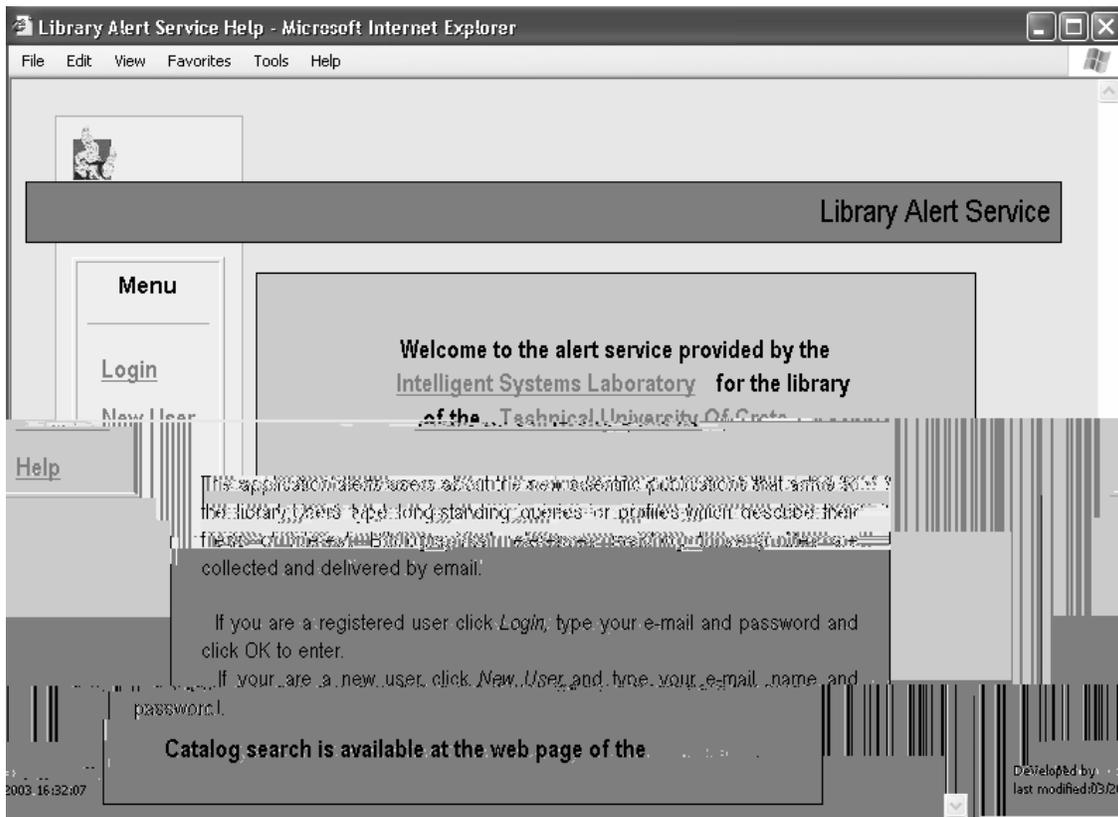


Figure 3.3-2 DLAlert: Welcome screen

In the 'New User Registration' screen (3.3-2) the user writes his e-mail address, his first/last name and the preferred frequency of notifications. A similar screen appears when a registered user wants to update his credentials.

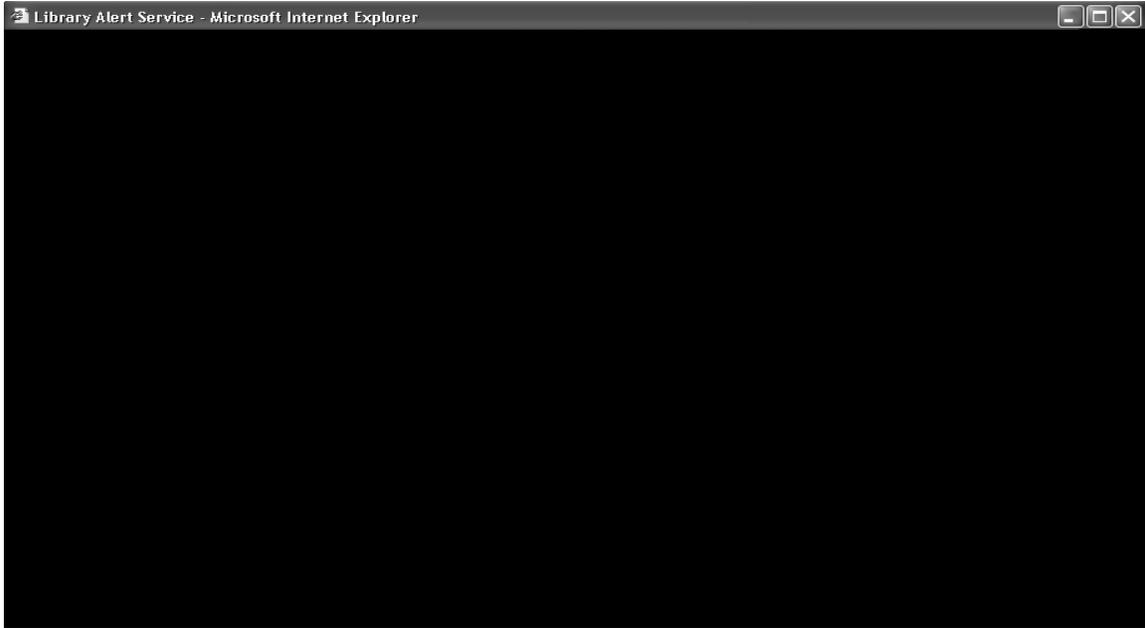


Figure 3.3-3 DLAlert: Registration form

After the registration to the service the user is expected to enter his profiles. The profiles define the user's fields of interest. In Figure 3.3-4 we see the empty account of a probably new user.



Figure 3.3-4 DLAlert: Empty account

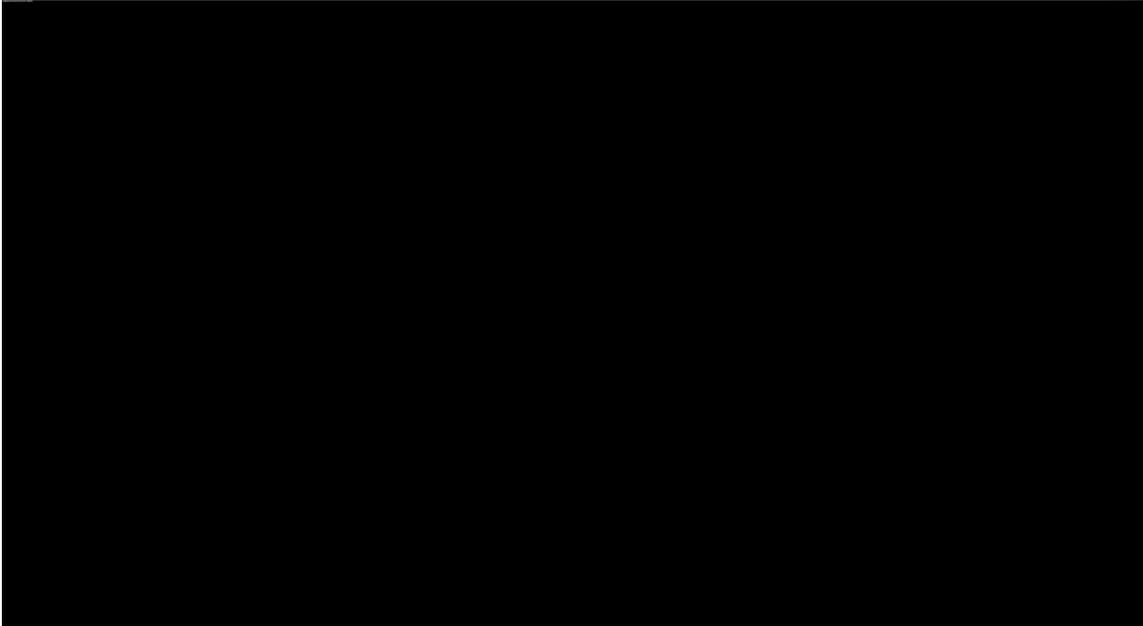


Figure 3.3-5 DLAlert: Profiles of the account

Suppose we enter a new profile. A form that allows us to type the profile name and queries on all bibliographical attributes appears (see Figure 3.3-6). Text queries based on a language (defined in next section) similar to the CTXRULE are allowed on sections: Title, Series, Author, Publisher, Subject and Notes. The publication year, ISBN and ISSN queries can contain only one number instead of keywords that will be found in the incoming publication. Operators and terms in queries are case insensitive. For a profile to be matched, all non-empty defined stored queries must be satisfied for a given document. The Profile description doesn't affect the matching publications of a profile. It is considered a phrase that reminds to the user the purpose of entering the given profile.

In Figure 3.3-6 we can see a sample profile with name "books about Programming". The profile contains two queries. The text query "programming or Java or C" requests documents that contain at least one of the words "programming", "Java" or "C" in the Subject bibliographical attribute. The query "2002" on the publication year field restricts the returned publications of the profile to those published in year 2002. So the desired document must contain at least one of the words entered in the text query and be published in the year 2002.

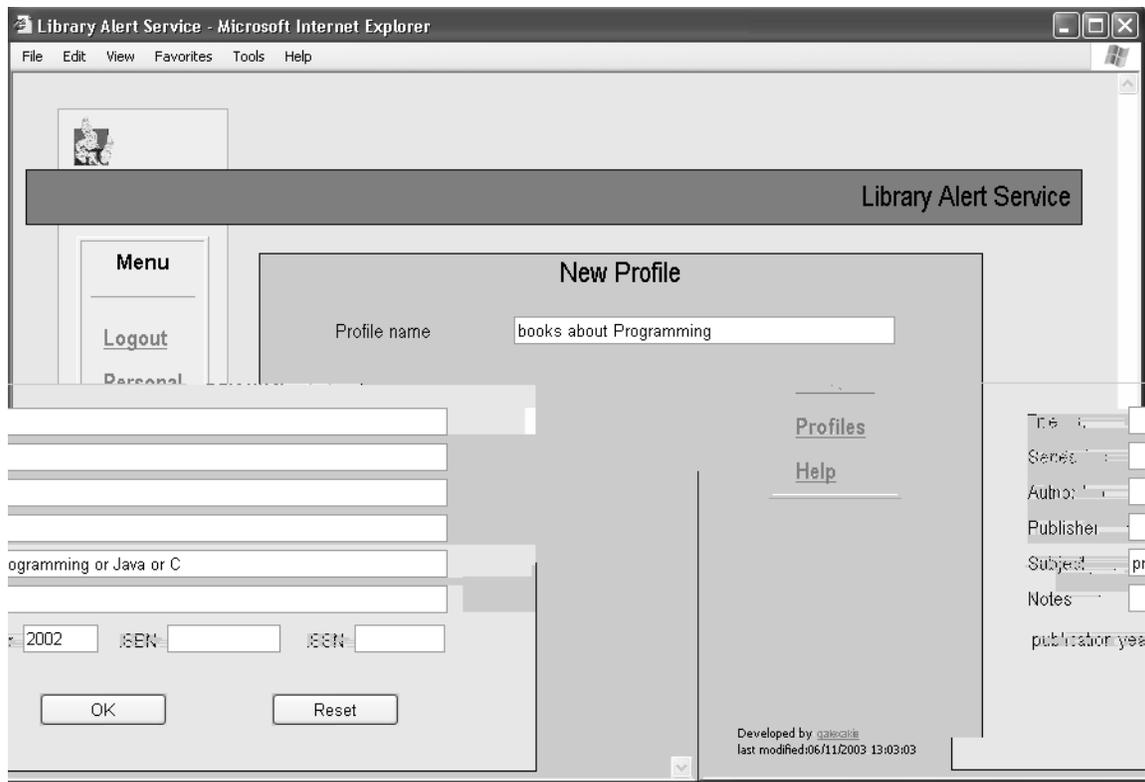


Figure 3.3-6 DLAlert: Sample profile

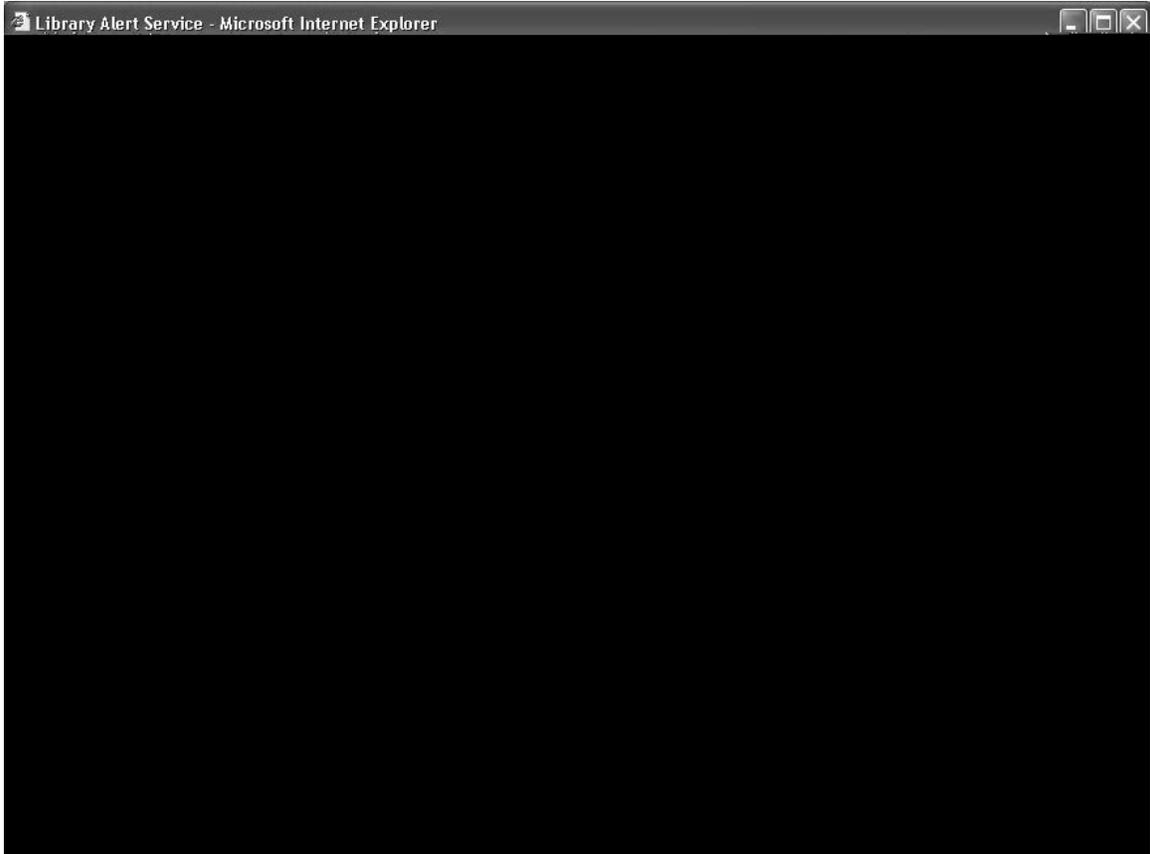


Figure 3.3-7 DLAlert: Wrong profile



Figure 3.3-8 DLAlert: Another profile

3.4 The language of the text queries

The language of the text queries (referencing the Title, Series, Author, Publisher, Subject, Notes bibliographical attributes) is a subset of the CTXRULE grammar (see Section 2.4.3). The available query types provided by the interface of DLAlert are shown in Table 3-4.1. Terms are considered words or phrases (series of tokens containing no operators). The queries are not case sensitive.

<i>term1</i>	documents that contain <i>term1</i> .	TITLE: mathematical programming
<i>term1 and term2</i>	documents that contain both terms.	TITLE: agents and (artificial intelligence)
<i>term1 or term2</i>	documents that contain <i>term1</i> or <i>term2</i> .	AUTHOR: (Bradley Brown) or Beck
<i>term1 not term2</i>	documents that contain <i>term1</i> but not <i>term2</i> .	NOTES: science not electronics
<i>near((term1,term2,...,termn),max)</i>	documents that contain all terms within a set of words with size <i>max</i> Order of terms is not specified. Limitation: <i>max</i> cannot be greater than 99.	TITLE: near((personal, computers) , 4)
<i>about(term)</i>	documents that contain concepts that are related to your query word or phrase. Limitation : <i>about(...)</i> and <i>near(...)</i> cannot occur both in the same field.	SUBJECT: about(engineering)
<i>\$term</i>	documents that contain words with the same linguistic root as <i>term</i> .	SUBJECT: \$library

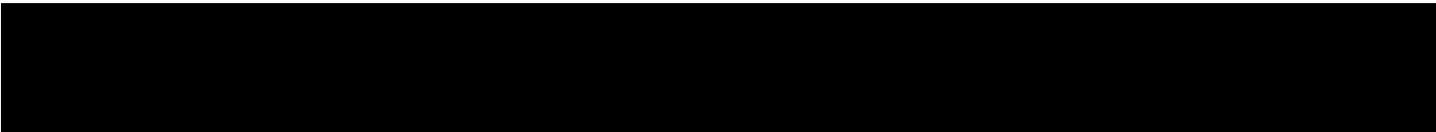
Category	Example
complex Boolean statements	TITLE: (mine or disasters) not industry
proximity and Boolean operators	TITLE: near((mine, disasters) , 5) and industry
concept queries and Boolean operators	SUBJECT: about(engineering) or software not databases
stemmed words and proximity	NOTES: software not near((advanced, \$electronics) , 3)

Table 3.4-2 Complex queries

Queries like those shown in table 3.4-3 are wrong and produce syntax error in case the user tries to enter one of them.

Error type

Example



Chapter 4

Database Schema

In this chapter we present in detail the design of the database schema of DLAlert. In addition to the basic requirements analysis and the Entity-Relation diagram we also deal with issues like primary-foreign key consistency and the necessary CTXRULE indices creation and maintenance (see also Section 2.4).

4.1 Requirements analysis

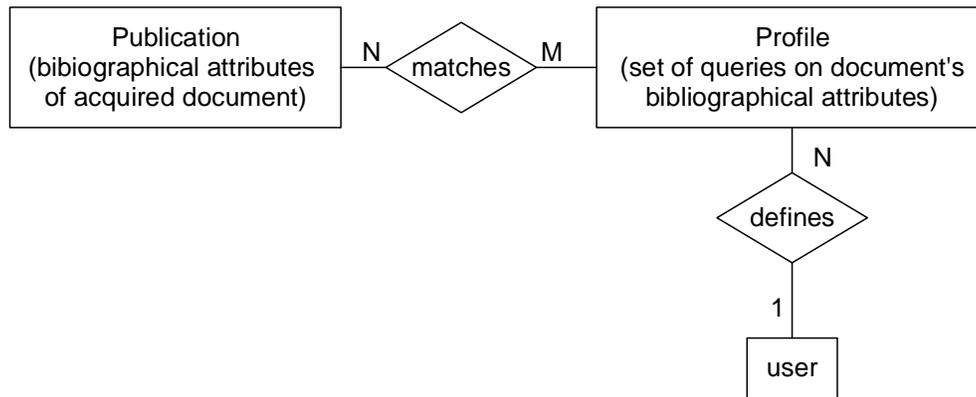


Figure 4.1-1 A high level Entity-Relationship diagram

The requirements of DLAlert for data resources involve only three entities (as shown in Figure 4.1-1).

- The entity user contains all necessary user credentials and information. These are the user's e-mail, first/last name and the password for his login into DLAlert. The request of password at user login ensures that all users have restricted access to their accounts. Another helpful feature is the desired frequency of notifications. DLAlert collects all matching documents of a certain user and sends a single message to him at the end of the desired interval ('DAY', 'WEEK', 'MONTH') containing all relevant bibliographical attributes. It is considered annoying to send a new e-mail message every time a new matching publication arrives.

- The entity publication contains all necessary bibliographical attributes of

bibliographical attributes Title, Series, Author, Publisher, Subject, Notes, Publication year, ISBN and ISSN. For a profile to be matched, all not null queries must be satisfied for a given document. A user can only access and define the profiles of his account. User also can specify a short string describing each profile. This description does affect the set of matching documents but allows a single user to define many profiles in a convenient way.

4.2 Relational schema

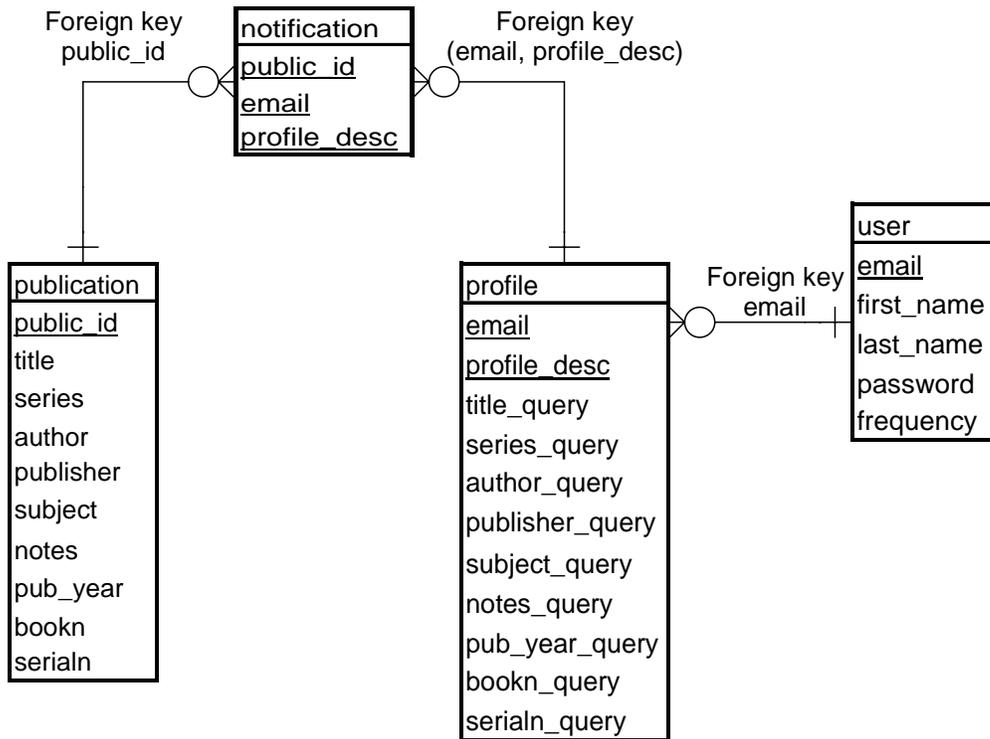


Figure 4.2-1 Relational schema

Considering the E-R diagram and the requirements analysis of Section 4.1, we construct the relational schema in Figure 4.2-1. The email address is unique for every user (primary key) and is also stored as foreign key in the table of profiles in order to restrict user access into his account. The primary key in profiles table consists of two columns (`email`, `profile_desc`) so that the profile description and the user's account email, uniquely identifies a profile. The primary key of publications table (`public_id`) is the local number of UNIMARC record (field 001) in TUC's Library (Sections 2.3.4 - 2.3.6).

In order to represent the relation “publication matches profile” (M to N) we declare a table (`notifications`) holding the necessary primary keys of the related tables. The primary key of table profiles (`email`, `profile_desc`) is considered foreign key for the notifications table. The filtering module populates this table with the primary keys of the satisfied profiles and matching documents. The notification module uses these records to send messages with the matching document’s bibliographical attributes.

All attributes are declared strings of variable length (data `varchar2` number) except those named `public_id`, `pub_year` and `pub_year_query` which are integers (data type number). The value of attribute frequency (`users`) can be either ‘DAY’, ‘YEAR’ or MONTH’ (*CHECK constraint*).

The tables of profiles and users are accessed through the web Graphical User Interface. The tables holding publication and notification records are populated and accessed only by stored procedures of the Oracle RDBMS.

4.3 Key consistency and atomic transactions

Suppose a user wants to update his email and his account contains profiles, and/or notifications on publications not send as messages yet. In other words the primary key value of a record, in users table must be changed, and this value is also stored as foreign key into another table (`profiles`, `notifications`). In all cases when a transaction references more than one inserts/ updates/ deletes and either all sub-operations must be committed successfully, or none of them, we consider this transaction atomic. The atomicity of actions like these is satisfied by using PL/SQL stored procedures. In our case we declared a package (named “transactions”) that handles updates of the `email` or `profile_desc` attributes (both part of foreign key). Also handles deletes on the `profiles`, `users` tables atomically. For example consider the scenario that user with email `del_email` unsubscribes from DLAlert. We execute the transaction using this procedure.

```
procedure delete_user( del_email varchar2 ) is
    pragma autonomous_transaction;
begin
    delete notifications where email=del_email;
    delete profiles where email=del_email;
    delete users where email=del_email;
    commit;
```

```
end delete_user;
```

When user unsubscribes, his data must be deleted from three tables user, profiles, notifications. The keyword "pragma autonomous_transaction;" ensures that either all three deletes in the PL/SQL block commit or none of them.

The specifications of other procedures of this package are

```
procedure update_user( old_email varchar2 ,  
                      new_email varchar2 );
```

Called if email value in a record, is changed from old_email to new_email. Declares the update of the e-mail value on the tables (profiles, notifications, users) as an atomic transaction.

```
procedure update_profile( user_email varchar2,  
                        old_desc varchar2,  
                        new_desc varchar2);
```

Called if profile description value in a record, is changed from old_desc to new_desc. Declares the update of the value on the tables (profiles, notifications) as an atomic transaction.

```
procedure delete_profile( user_email varchar2,  
                        del_desc varchar2);
```

Called if profile with primary key value (user_email, del_desc), is deleted. Declares the deletion of a profile on the tables (profiles, notifications) as an atomic transaction.

4.4 Indexing of the stored queries

The next step is to declare columns that contain stored queries and create the necessary CTXRULE indices. As shown in Section 2.4.4, we must first index the queries in order to be able to collect matching profiles. The bibliographical attributes (Title, Series, Author, Publisher, Subject, Notes) contain plain text and the corresponding stored queries are generated using the CTXRULE grammar and reference these sections. The queries on ISBN, ISSN are ten and eight digit strings respectively (0-9 or X). Many records of TUC's Digital Library have multiple ISBN and/or ISSN numbers and

the number included in the query must be equal to one them. A solution to this problem was to index also queries on ISBN, ISSN numbers. The *MATCHES* operator is able find matching publications with multiple book/serial identifiers. The publication year attribute has always a single value and text queries on this attribute, using the CTXRULE grammar, are usually meaningless. Instead of using the *MATCHES* predicate to evaluate satisfied queries on publication year, we use the standard SQL statement ('=') of equality in a *WHERE* clause. So we have to create eight indices on the columns containing stored queries (all except `pub_year_query`) that use the CTXRULE functionality.

As we showed in Section 2.4.4 indexing null queries is produces index errors. Given the structure of DLAlert and the queries we want to provide, we do not expect the user to describe every attribute of the requested document but at least one. So we have to choose a non reserved symbol to be inserted in0.4042in fields that the user does not specify a condition. We choose this symbol to be the less than symbol (“<”) because is not associated with any functionality or operator. Also this character is skipped by the indexing engine so the number of columns containing this symbol does not affect the size of the index. We construct a simple trigger that is executed before every insert or update on the table profiles and inserts the symbol “<” in0.4042of null. This trigger has `if...then` rules for every stored query like this.

```
if ( :new.title_query is NULL ) then
    :new.title_query:='<';
end if;
```

Queries that contain only the symbol “<” are displayed as empty strings on the Graphical User interface (see Sections 6.4 - 6.5) so that this mechanism is transparent to the user.

In order to ensure that filtering results are correct and consistent with the queries, the indices should be synchronized before filtering with the base table after DML actions. For this purpose we declared the package “indexing” with a procedure (“sync”) that synchronizes all CTXRULE indices sequentially before any execution of the filtering module. In DLAlert we assume that index synchronization and filtering are executed at least one time at the end of the day.

4.5 Conclusions

In the last chapter we analyzed the requirements that the database used by DLAlert must meet. We explained in detail the database schema and preceded on to particular design issues (atomic transactions and index creation). In the next section we present the main PL/SQL packages of the system: the filtering and notifying module.

Chapter 5

PL/SQL packages

In this chapter we present the filtering and notification modules of DLAlert implemented as PL/SQL packages. Packages are constructs that allow us to logically group procedures, functions, object types, and items into a single database object. PL/SQL package have similar functionality as classes in object oriented languages (Java or C++), except every package is instantiated only once during a database session.

5.1 Filtering module

Suppose we have the table `publications` in the database schema of Figure 4.2-1 populated with bibliographic attributes of documents and the profiles table `withqueries` of the CTXRULE grammar. In order to be able to produce the necessary messages, we must first find the matched profiles for every document. A profile is matched if all set conditions on the corresponding attributes of the document are satisfied.

5.1.1 The algorwitm

As we have shown in Section 2.4.4 to find all matched profiles for a single document we use the following algorwitm. `current_publication` is a parameter for the procedure `find_matched_profiles`.

```
procedure find_matched_profiles
  (current_publication publications%rowtype) is
begin
  for matched_profile in
    (
      SELECT <needed columns>
      FROM   <table holding the profiles>
      ...
      MATCHES condition(s)
    )
  loop
    ACTION EXECUTED FOR EVERY matched_profile
  end loop;
end;
```

First we have to construct the appropriate *SELECT* statement that collects all matching profiles. If the variable holding the given publication is named `current_publication` and we collect all matched profiles according to the title attribute and corresponding query we have

```
select email,profile_desc from profiles
where matches(title_query, current_publication.title)>0
```

Suppose now we want find profiles that satisfy not only the query on the attribute `title`

There are only two possible solutions to our problem. The first one is treating results of *SELECT* clauses as sets and using the intersection of the intermediate results of the partially satisfied profiles. This can be done as follows:

```
(select email,profile_desc from profiles
 matches(title_query, current_publication.title) >0 )

intersect

(select email,profile_desc from profiles
 matches(title_query, current_publication.author) >0 )
...
...
intersect
(select email,profile_desc from profiles
 matches(serialn_query, current_publication.serialn) >0 )
```

Another solution is using the intermediate results and issuing over them a *SELECT* clause that ensures their primary keys' equality. This can be done as follows:

```
select Title_Results.email,Title_Results.profile_desc from

  ( select email,profile_desc from profiles
    where matches(title_query,current_publication.title)>0
  ) Title_Results ,

  ( select email,profile_desc from profiles
    where matches(series_query,current_publication.series)>0
  ) Series_Results ,
...
...
...
  ( select email,profile_desc from profiles
    where matches(serialn_query,current_publication.serialn)>0
  ) Serialn_Results

where
  and Title_Results.email=Series_Results.email
  and Title_Results.profile_desc=Series_Results.profile_desc
...
...
...
  and Title_Results.email=Serialn_Results.email
  and Title_Results.profile_desc=Serialn_Results.profile_desc
```

Both of the SQL statements result the same performance measures and are processed by the Oracle SQL query processor in a similar way.

In Section 4.4 we explained that null fields produce index errors so we choose to use the symbol '<' instead of empty query cells. The symbol '<' is skipped by the Lexer during the indexing process so it does not appear in the indices. So the intermediate *SELECT* clause of the profiles, that match a single attribute in the previous SQL statements, taking in account the cells described as null queries is (for example the *title_query* with the title attribute) are implemented as follows.

```
(select email,profile_desc from profiles
  where matches(title_query, current_publication.title)>0)
union
(select email,profile_desc from profiles
  where title_query = '<')
```

The action executed for every *matched_profile* is an insert of the primary keys of the matched profile and relevant publication into the notifications table.

```
insert into notifications(public_id,email,profile_desc)
  values( current_publication.public_id,
         matched_profile.email,
         matched_profile.profile_desc
        );
```

We also do not forget to issue the command `commit;` so the transaction is committed. A `commit;` statement ends a transaction and makes permanent any changes performed. This statement is preferably issued outside the `for...loop` and executed once as a commit's response time is fairly flat, regardless of the transaction size.

To find matching profiles for all publications another `for...loop` is needed to call the previous procedure on all documents.

```
for all_documents in
(
  select * from publications;
)
loop
  find_matched_profiles (all_documents)
end loop;
```

5.2 Notifying module

Once the notifications table is populated with matching profiles and publications, all we have to do is to summarize the matched documents for every user and send him a single e-mail. The preferred frequency of notifications does not affect the filtering process but all new publications are filtered against all profiles but the frequency controls the time the e-mail message will be sent. For sending e-mail messages from the database we use supplied package UTL_SMTP. We first present the basic features of this package and then we describe the whole functionality of our module.

5.2.1 The UTL_SMTP package

SMTP [43] stands for Simple Mail Transfer Protocol. This is the protocol that was developed to allow people around the world to exchange electronic mail. UTL_SMTP [30, 33, 35, 40] is an email utility that provides us with the ability to email from the database. In other words, we can dynamically generate email from the database and we can dynamically send it to different people based on different criteria. The message constructed can be sent as a standard ASCII text email or an enhanced HTML email.

A SMTP connection is initiated by a call to `open_connection`, which returns a SMTP connection. After a connection is established, the following procedure calls are required to send a mail (we do not specify the complete syntax):

- `helo()` or `ehlo()` - identify the domain of the sender
- `mail()` - start a mail, specify the sender
- `rcpt()` - specify the recipient
- `open_data()` - start the mail body
- `write_data()` - write the mail header/body (multiple calls)
- `close_data()` - close the mail body and send the mail
- `quit()` - close the SMTP connection

Using these commands we define a rather complex PL/SQL procedure inside the notifying module's package which has the following syntax.

```
procedure send_mail
    (in_mail_server,      in_sender_email,
    in_recipient_email,  in_recipient_name,
    in_html_flg,         in_subject,
    in_importance,       in_body);
```

`in_mail_server` is the mail server ,

default value 'intellix.intelligence.tuc.gr'.

`in_sender_email` is the senders mail address,

default value 'alrererererere317 0 TD0001 er

```
Text := htf.fontOpen( cface => 'Arial Narrow', csize => '5')
      || 'T.U.C Library Alert'
      || htf.fontClose
```

Assigns to the string `Text` the value :

```
<font face="Arial Narrow" size="5"> T.U.C Library Alert </font>
```

We do not present in detail the functions used, because generating HTML from PL/SQL code is a rather complex issue. The algorithm that collects all matched publications for a given user, and generates a message is.

```
procedure notify_user( user_email varchar2 ) is
begin
--GENERATING THE HEADER OF THE MESSAGE

for matched_publication in(
        select publications.*
        from publications,
            (select public_id
             from notifications
             where email=user_email
             group by public_id) matched
        where matched.public_id=publications.public_id
    )
loop
    --APPEND ALL NON-EMPTY BIBLIOGRAPHICAL ATTRIBUTES
    --TO THE MESSAGE FOR EVERY matched_publication
end loop;

--FINISHING AND SENDING THE MESSAGE
--USING THE send_mail PROCEDURE.
end;
```

`user_email` is a input string containing e-mail of the user to notified.

We concentrate on the SQL *SELECT* clause used.

The clause

```
(select public_id
 from notifications
 where email=user_email
 group by public_id) matched
```

retrieves all matched publications of the user with `email=user_email` and removes ,duplicate matched entries from the result, in case the user has more than one profiles matching the same publication. Also names the results as table `matched`.

The clause

```
select publications.*
from publications,matched
where matched.public_id=publications.public_id
```

Retrieves the matched publications for a single user, from the table holding the bibliographical attributes, using the primary keys of the matched ones collected in table `matched`.

In order to produce and send messages to all users that have matched profiles we use the algorithm.

```
procedure notify_all is
begin
for current_user in
(
select email from notifications group by email
)
loop

    notify_user(current_user.email);
delete from notifications where
    email = current_user.email;

end loop;

commit;
end;
```

The *SELECT* clause retrieves all unique e-mail addresses of the users that have matched profiles in the notifications table, and calls the `notify_user` procedure for each one of them.

We do not forget to delete the sent notifications for each successfully sent message with the simple DML statement.

```
delete from notifications where
    email = current_user.email;
```

If we want to notify users according to their desired frequency of notifications we must construct the appropriate procedures that notify all users that have defined the same interval between e-mail messages. For example if we want to notify all users that want to be sent e-mail messages every day, in case there is a matching publication, instead of the previous SELECT statement that controls the for...loop we issue.

```
select users.email
from notifications, users
  where users.frequency='DAY'
  and users.email=notifications.email
group by users.email
```

Therefore we send messages to each category of users separately. For example the group of users that selected day as preferred frequency, are notified each day, those who preferred week once a week etc. As we said in the previous section the profiles are not filtered separately but the frequency of notifications affects only the time the messages will be sent to the user. In Chapter 9 we explain in more detail the scheduling of each module.

5.3 Performance

We tried to measure the performance of DLAlert. Our goal was to ensure that the time needed for filtering would be acceptable if we consider that this process would be executed once a day. Our measures represent the worst case scenario on the server of the Intelligent Systems Laboratory (two Pentium III processors, 1 GB RAM). We took documents from the work of Theodoros Koutris and Christos Tryfonopoulos on a DIAS implementation [53,54]. We also generated profiles on random keywords encountered in those documents, using the profile generator of DIAS of the same implementation (only Boolean and proximity statements) and parsed them into equivalent CTXRULE expressions.

The time taken for indexing 340082 complex stored queries (that is 100000 profiles with 1 to 4 stored queries on attributes each) in the server of the Intelligent Systems Laboratory was approximately 10 minutes. We do not measure the time taken to insert the profiles. This was considered a quite satisfactory performance measure since indexing 340082 stored queries means that there are 340082 new queries (inserts or updates) since the last index synchronization (last day for example) which is most unlikely to happen even for the popular alerting applications we described in Section 2.1.

Assuming that index synchronization is executed once a day, we are able to store an even larger amount of profiles. The time taken for roughly the one tenth of the queries (3300), was approximately 2 minutes which means that time required is not a linear function of the profiles inserted.

As stated by on the “*Oracle Text Technical Overview*” [27], CTXRULE query performance depends mainly on the size and number documents. As these factors increase, there are more unique words, each of which results in a query on the index. Performance is also affected by number of unique rules indexed and the complexity of stored queries. However, the number of unique rules has much less impact on query performance than size of the document.

The SQL Query that collects matching profiles is rather complex in both cases. The time taken for filtering 84 documents against 340082 stored queries (that is 100000 profiles with 1 to 4 stored queries on attributes each) in the server of the Intelligence Laboratory was approximately 13 minutes in both of the previous implementations. The size of an attribute of a document could vary from a small amount of to a few thousands of words. The total size of all documents (which is the most important factor) was 3.5 Mbytes (approximately 380.000 words) which is considered a very large amount of words for filtering. In the actual implementation of DLAlert when records are retrieved from the Digital Library, the average record size is much smaller, since bibliographic attributes are short strings usually. Even assuming that filtering, which is executed once a day requires roughly 13 minutes on average, this time is fairly acceptable for the purposes of our application. Oracle states that the expected response time for filtering 64337 words against 16000 indexed queries is approximately 20 sec. In Chapter 9 we present techniques, proposed for future work on DLAlert that will reduce this time further.

We have to underline that filtering using *MATCHES* is always a CPU time consuming task and in the server we used for development, there many other processes running all the time. Also we did not utilize any additional functionality provided by Oracle that speeds up the overall database performance. However our goal was to roughly estimate the time needed for filtering and indexing, given the usual workload on the server of the Intelligence Laboratory and decide whether DLAlert could be deployed on this computer. Our measures do not evaluate the overall performance of the Oracle Text.

5.4 Conclusions

We explained in detail the essential PL/SQL components of DLAlert, the filtering and the notifying module. The filtering module finds all matched profiles for every publication and stores the primary keys of those rows in a table. The notifying module processes those data, produces and sends dynamic HTML e-mail messages to each user containing the bibliographical attributes of all matching publications. In the next chapter we present, the Graphical User Interface of DLAlert.

Chapter 6

The Graphical User Interface

In this chapter we present the GUI of DLAlert. We start with an overview of this component and continue with particular technical issues and implementation details. The URL of DLAlert is <http://www.intelligence.tuc.gr/alert/login.html> .

6.1 Middle application tier architecture

In this section we explain in detailed the 3-tiered architecture of our web application,

- The Web tier generates presentation logic, accepts user input from HTML and generates appropriate responses for the user. We implement this tier as pages created with *Java Server Pages* (JSP) [45, 46] technology on the application server. JSP's simplify the development of dynamic Web pages. JSP technology enables us to mix regular, static HTML with dynamically generated content. The parts that are generated dynamically are marked with special HTML-like tags and contain Java code.

Apart from the standard JSP tags, in our application we used the *Oracle9iAS Containers for J2EE* (OC4J) Custom Tag Library for SQL, provided by Oracle with the Oracle9i Application Server [40, 47]. A tag library defines a collection of custom actions for JavaServer Pages. OC4J is a framework for rapid JSP development. OC4J tags related to database access, support functionality for opening/closing a database connection, executing a query or any other SQL statement (DML or DDL) within JSP code. Except the standard dynamic pages related to presentation, we have constructed JSP's, not visible to the user, that process transactions on user credentials and already validated profiles. Although OC4J functionality is provided with the Oracle AS, the applications developed with this framework, can be deployed into any other application server that supports JavaServer Pages technology.

- The Business Tier implements business logic related to the user's session and is developed using *Enterprise JavaBeans* (EJB) Technology. An EJB is a server-side web component, written in Java that encapsulates the business logic of an application. In DLAlert this functionality includes user authentication, profile validation and data retrieval (user credentials and profiles) from the database. Also atomic transactions that update/delete foreign keys and require stored procedures calls (see package 'transactions' Section 4.3) are handled by the EJB. A stateful session bean is an EJB
-

JavaCC generates source code for parsers using LL(k) grammars. We explain in detail the implementation of syntax checking in Section 6.4.

We could have also included all transaction handling functionality inside the EJB instead of using the OC4J custom tag library, but applications that use this framework can easier be developed and maintained. However the JSP's interact with the EJB in a way that ensures security and isolation of the user's session.

- The Web and Business Tier communicate with each other using *Remote Method Invocation* (RMI). RMI is a Java based Application Programming Interface (API) for *distributed object computing* and Web connectivity. RMI allows an application to obtain a reference to an object that exists elsewhere on the network but then invoke methods on that object as though it existed locally. So, the web and business tiers can be implemented in different J2EE platforms, although in our case they are deployed in the same application server.
- The Middle Application Tier communicates with the database using the *Java Database Connectivity* interface (JDBC) [35, 38, 40]. JDBC API is a specification for database connectivity using Java. Software vendors (like Oracle) produce their own JDBC drivers that implement the API specification in a greater or lesser degree, but all of them support a common set of interfaces. Thus the way the programmer interacts with the database, is to some extent independent to the JDBC driver used.
- The Database (also called EIS: *Enterprise Information System*) tier includes the RDBMS infrastructure (both data and stored procedures). We have already explained in detail the role of the RDBMS in Chapter 3.

6.2 The Enterprise Java Bean

In the next sections we concentrate on the business logic of DLAlert. First we present the main class used as an Enterprise Java Bean.

The class of the EJB is called 'LoginBean' and maintains login and account information of the user session. The main private fields of this class are:

JDBC related fields.

- o `java.sql.Connection conn` the JDBC connection to the database.
- o `java.sql.Statement stmt` the SQL statement to executed

Database schema related fields.

- o `java.lang.String dbUser` the Username for database schema (constant)
-

- o `java.lang.String dbPass` the Password for the database schema (constant)
- o `java.lang.String dbURL` the URL of the database (constant)

Account related fields.

- o `java.lang.String username` the username of the account (e-mail)
- o `java.lang.String password` the password for the account
- o `java.lang.String[] ProfileArray`

An array of strings with the profile descriptions of all profiles inside the account

Objects representing entities inside the database.

- o `ReadProfile ResultProfile`

Object representing the profile to be inserted/updated or the profile read from the database. This object holds all the queries of the profile class in a complex it is a result of a query.

Profile validation related methods (explained in Section 6.6).

o `ReadProfile getReadProfile(java.lang.String ProfileDesc)`

Returns an object holding all queries of the profile with name `ProfileDesc`. Calls the constructor of `ReadProfile` class.

o `StoreProfile getStoreProfile()`

Returns a profile to be stored, already parsed.

o `StoreProfile getStoreProfile(...)`

Constructs, parses and returns the parsed profile to be stored as an object. Calls the constructor of `StoreProfile` class.

Methods that prevent primary key constraint violation error.

o `Boolean ProfileExists(java.lang.String NewProfileName)`

Returns true if profile with description `NewProfileName` already exists inside the user's account. Prevents primary key constraint violation on the table profiles.

o `Boolean UserExists(java.lang.String NewEmail)`

Returns true if user with e-mail `NewEmail` already exists. Checks before new user registration/credentials update.

Methods that represent atomic transactions (see Section 4.3) – call PL/SQL stored procedures of package 'transactions'.

o `void DeleteProfile(java.lang.String DelProfileName)`

Deletes a profile inside the account of the user with name `DelProfileName`.

o `void DeleteUser()`

Deletes all user information from the database - unsubscription

o `void UpdateUser(java.lang.String NewEmail)`

Updates current user's e-mail to `NewEmail`.

o `void UpdateProfileDesc(java.lang.String OldProfileName,
java.lang.String NewProfileName)`

Updates profile name of profile `OldProfileName` inside the account with profile `NewProfileName`

6.3 OC4J custom tag library

Transactions can be declared inside JavaServer Pages. These transactions use OC4J custom tag library for SQL functionality.

The tags used from this library are:

We use the `dbOpen` tag to open a database connection for subsequent SQL operations:

```
<database:dbOpen
  [ connId = "connection_id" ]
  [ scope = "page" | "request" | "scope" | "application" ]
  user = "username"
  password = "password"
  URL = "databaseURL"
  [ commitOnClose = "true" | "false" ] >
... OPTIONALLY JSP CODE ...
</database:dbOpen>
```

Parameters :

- o `connId` -- Optionally used to specify an ID name for the connection. You can then reference this ID in subsequent tags such as `dbExecute`. Alternatively, we can nest `dbExecute` tags inside the `dbOpen` tag.
- o `scope` (used only with a `connId`) – We use this to specify the desired scope of the connection instance. The default is page scope.
- o `user` – the username of the database schema.
- o `password` – the password for the database schema.
- o `URL` – the URL of the RDBMS.
- o `commitOnClose` -- "true" for an automatic SQL commit when the connection is closed or goes out of scope. The default setting is "false" for automatic rollback on connection close.

We use the `dbExecute` tag to execute a single DDL or DML statement inside the tag `dbOpen` or outside of it using the same `connId` and `scope` parameters. The syntax for this tag is.

or missing operator.

- Queries that contain obvious syntax errors like unclosed parentheses, missing term

grammar. For each of the cases bellow, an index error appears.

queries, else an error message appears. There are several restrictions on the CTXRULE

database but rejected by the web interface. The user is not allowed to define wrong

according to the CTXRULE grammar. The invalid profiles should not be inserted into the

bibliographical attributes (Title, Series, Publisher, Subject, Notes) are generated

6.4 Preventing CTXRULE index errors

We must introduce the mechanism that validates profiles. The text queries on

Before explaining in detail the components that store/read profiles from the database

missing parenthesis

Term2 missing

Term1, missing

operator between *term1* RDBMS and

(information systems

security and

RDBMS name (data, warehousing),6)

RDBMS name (

- Queries that violate limitations on certain operators.

Rest of reserved operators

Operator	Symbol	Meaning
AND	&	Boolean and
OR		Boolean or
NOT	~	Boolean and-not
NEAR	;	proximity

Operator	Symbol	Meaning
(none)	\$	stem
ABOUT	(none)	related concepts
(none)	()	grouping characters
(none)	[]	grouping characters

We have decided to use only word operators when possible (AND, OR, NOT, NEAR). Also we do not use '[']' as grouping characters. The stemming character (\$) is the only symbol operator used.

Therefore analyzing the requirements of our application we conclude that we must implement a mechanism that escapes or rejects the following reserved words and symbols.

- Escaped reserved words : ACCUM, BT, BTG, BTI, BTP, FUZZY, HASPATH, INPATH, MINUS, NT, NTG, NTI, NTP, PT, RT, SQE, SYN, TR, TRSYN, TT, WITHIN .
- Escaped symbols: &, ? , - , ; , ~ , > , * , %.
- Any other special character or symbol is omitted.

The CTXRULE index contains only keywords and escaped symbols are never indexed by default. Therefore including an escaped special symbol in a query does not affect the filtering results. Special symbols are usually treated as token delimiters by the index engine by default.

We could not expect the user to be an expert on the CTXRULE language so we must construct a parser that automatically escapes reserved tokens. This functionality should not be visible to the user so that characters { } \ added by the parser are not visible from the web GUI.

➤ Empty Queries.

As we said empty cells in profiles are substituted by the symbol <. This character is skipped by the indexing engine so it is not included in the index. This symbol also should not be visible from the web GUI.

As a conclusion, we have constructed a parser that is executed before storing profiles inside the database:

- Checks queries for syntax errors.
- Allows only two digits on the proximity parameter (< 100).
- Allows only one of the statements *about(...)* or *near(...)* in the same text query.
- Turns themes inside *about(...)* clauses to lower-case.
- Escapes reserved words and symbols.

To implement such functionality we used JavaCC, a compiler generator for Java. JavaCC processes a text file that defines the grammar and the semantic actions of the compiler, and generates the appropriate source Java code. To construct this parser we have used the following LL(1) grammar. We must underline that this grammar does not define the actual CTXRULE

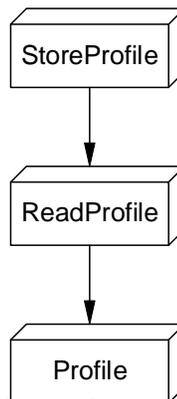
(6) *group left or _exp right*

A group of expressions starts and ends with parentheses.

(7)

must implement an object that holds the values of the profiles. The session E
not be to provide such functionality because the wrong query will be lost. Therefore we
Information Alert System for Digital Libraries - 86 -
the form is always updated with data from the RDBMS, in case of syntax error we will

contains an invalid query, the application should be able to point out the syntax error. If
6.5 Parsing the text queries
profiles and produces error messages. If a user tries to insert or update a profile that
As a conclusion to the previous section, we need a mechanism that parses the



- The class `Profile` is an abstract class and cannot be instantiated.

The arrows represent an “is a” relation.

Figure 6.5-1 Class Hierarchy
For this purpose we have constructed a class hierarchy as shown in Figure 6.6-1.

➤ The class `StoreProfile` includes all fields and methods of `ReadProfile`. A `StoreProfile` object is instantiated with fields containing parsed text queries, which are defined by the user. The constructor method of `StoreProfile` calls the JavaCC generated parser which performs the necessary semantic actions on all text query fields, before assigning the strings to the private variables. Among the methods inherited from `ReadProfile` the class includes the following ones.

- `public Boolean isValid(int QueryIndex)`
Returns true if the referenced text query is valid. `QueryIndex` defines which text query of the profile is referenced.
- `public String getErrorMessage(int QueryIndex)`
Returns the string “Syntax Error” displayed on screen, if the referenced text query is invalid.
- `public String getHeaderMessage()`
Returns the string “Encountered XX wrong queries ...” displayed on screen, if the referenced profile contains wrong query.
- `public int NumberOfErrors()`
Returns the number of wrong queries of the profile.

This class hierarchy allows us use call two different constructors virtually for the same `Profile` object (`ReadProfile()` and `StoreProfile()`) according to the source of text queries assigned. The constructor `StoreProfile()` calls the parser and the constructor `ReadProfile()` just assigns the text queries to the fields. Thus we avoid re-parsing of already parsed text queries.

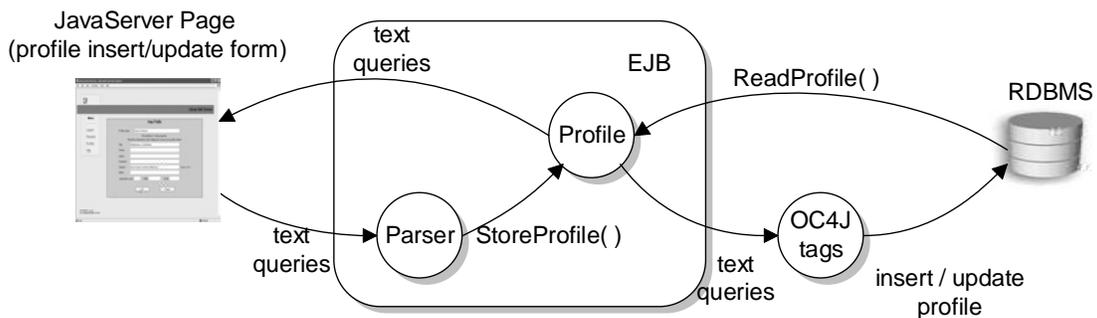


Figure 6.5-2 Profile insert / update mechanism

The functionality that performs inserts/updates on profiles is shown in the schema above and can handle the following four actions in any valid sequence.

- If a new profile is to be defined the JavaServer Page is fields with empty strings.
 - If a profile is to be updated, the form is filled with the actual RDBMS data. The EJB calls `ReadProfile()` constructor and the JavaServer Page reads the text queries from the object (omitting escape characters `{ } \` and one character strings with the symbol '`<`', displayed as empty field).
 - If the user enters a valid profile to be inserted / updated the EJB calls the `StoreProfile()` constructor, parses the profile, the JSP with the OC4J tags reads the values from the object `Profile` and performs the transaction
 - If the user enters a wrong profile the EJB calls the `StoreProfile()` constructor, parses the profile and the form containing the wrong queries and the errors messages appears. In this case the transaction is not performed.
-

Chapter 7

The Observer

In this chapter we present the mechanism that collects records from Digital Libraries

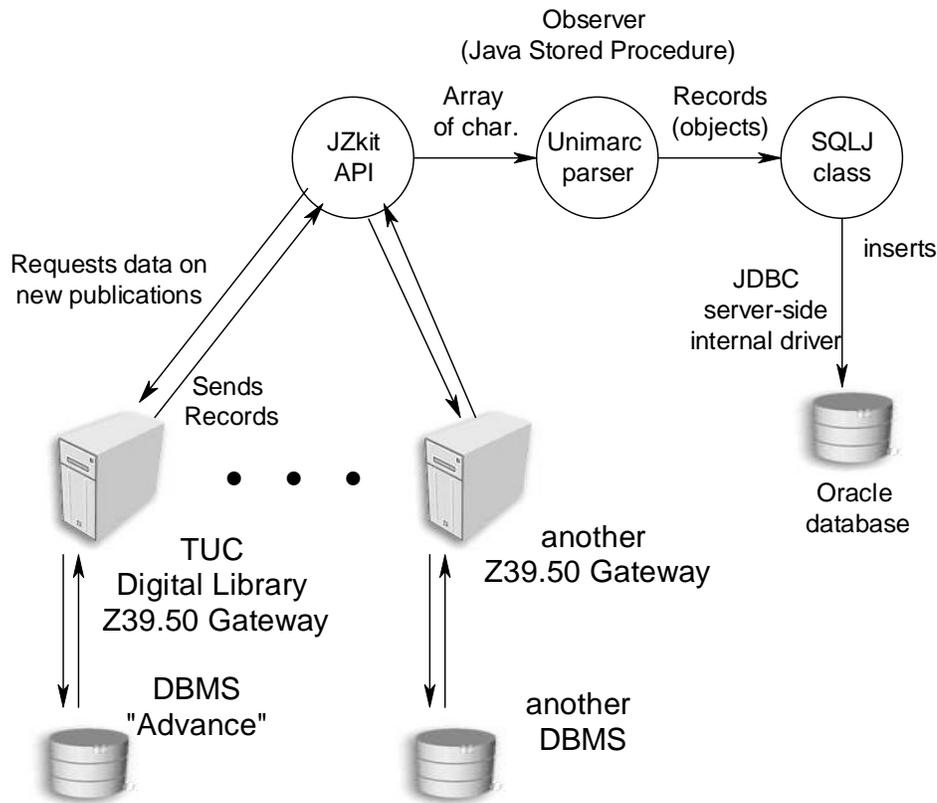


Figure 7.2-1 Architecture of the Observer

- JZKit [15] is an open source Java toolkit for building distributed information retrieval systems, with particular focus on the Z39.50 Information Retrieval standard. JZkit offers us functionality that helps us develop clients for the Z39.50 protocol. We have already presented the Z39.50 main facilities and services in Section 2.3, in this chapter we focus on the particular characteristics of the Observer. The code developed writes requested records in an array of characters, processed by the next component.
- The UNIMARC [17] parser is a typical parser generated by JavaCC [50]. This parser processes UNIMARC records as input and maps the UNIMARC fields to the desired bibliographical attributes (Title, Series, Author, Publisher, Subject, Notes, Pub. Year, ISBN, ISSN). The UNIMARC format was presented in detail at Section 2.3.5. This component virtually processes structured text and produces a set of objects

Using queries like the above at the end of each month we can retrieve the necessary records. The complete algorithm developed utilizes the Initialize, Search and Present services. We must mention that the target cannot return all requested records in one response (response limited by the segmentation service) so we have to count the records returned until all of them are retrieved. Phrases enclosed in < > represent variables.

```
Initialize ( URL : dias.library.tuc.gr , port : 210 )
    returns initialization parameters
Search ( database-name : "Advance" ,
    query : @attrset bib-1 @attr 1=32 <current year>-<month ntigreek> )
    returns <total number of records>
<starting point> = 0

repeat
    Present ( number of records , starting point )
        returns <number of returned records> , <records in UNIMARC>
    <starting point> += <number of returned records>
    write <records in UNIMARC> in an array of char.
until <starting point> equals < total number of records>

terminate connection
```

The previous algorithm Initializes a Z-association with the target, issues a query and sends requests until all records are returned. The records of the Present response are written in an Array in order to be processed further by the next module.

7.4iUNIMARC parser

The UNIMARC record format is not supported at the time by Oracle Text. Thus we have to parse the incoming records into plain text in order to insert them in the database schema explained in Chapter 4. The parser processes the UNIMARC records and maps UNIMARC fields into the fields used by DLAlert (Title, Series, Author, Publisher, Subject, Notes, Pub. Year, ISBN, ISSN). The parser maps the UNIMARC fields to those bibliographical attributes according to the Table 7.4-1.

Destination	UNIMARC fields
Local Number	001
Title	200,5XX,4XX except 410
Series Title	410
Author	7XX
Publisher	210
Subject	60X
Notes	3XX
Publication Year	210 \$d
ISBN	010 \$a
ISSN	011 \$a

Table 7.4-1 UNIMARC field mapping

Local number is the unique identifier of the record inside the database of the Digital Library of TUC. We use this number as a primary key for our schema (public_id). Other fields included in the UNIMARC record (like information about the book's lending) are omitted.

For example consider the following record. The extraction of bibliographical attributes is shown in the Figure 7.4-1. The date of acquisition is not inserted in the Oracle database.

001 TUCb10024364	←	Local Number
003 TUC		
005 19990705122200.0	←	ISBN
010 -- \$a0844235172 \$b (recyclable paper)		
019 -- \$a94016173		
100 -- \$a19940404d1995 0grey0105 ba		
101 1- \$a eng		
200 2- \$a The international business book \$f Vincent Guy, John Mattock	←	Title
210 -- \$a Lincolnwood, Ill., USA \$c NTC Business Books \$d 1995	←	Publisher, Publication Year
215 -- \$ax, 178 p. \$c ill. \$d 23 cm.		
300 -- \$a "All the tools, tactics, and tips you need for doing business across cultures"--Cover.		Notes
320 -- \$a Includes bibliographical references (p. [171]-173) and index.		
606 6- \$a International business enterprises \$x Management	←	Subject
676 -- \$a658/.049 \$v 20		
680 -- \$aHD62.4 \$b .G89 1995		
700 -0 \$a Guy \$b Vincent		Author
701 -0 \$a Mattock \$b John		
709 -- \$a NTC Business Books		
801 -0 \$a GR \$b TUC \$g AACR2		
960 -- \$a 1999-10ΥΝΙΟΣ	←	Access point (date of acquisition)
970 -- \$a ΝΤΟΥΝΤΟΥΝΑΚΗ \$b ΧΑΡΑ \$z 1999-07-05		
852 -- \$b ΠΟΛΚΡ \$b ΒΙΒΛΙΟΘΗΚΗ \$b ΚΥΣ \$h HD62.4.G89 1995 \$m 41011 \$p 00141011 \$t 1 \$y Στο ραφι		

Figure 7.4-1 Sample UNIMARC record

The fields of the processed record are shown above (represent private variables of the object LibraryRecord).

public_id	title	series	author	publisher
10024364	The international business book Vincent Guy, John Mattock	<null>	Guy Vincent Mattock John NTC Business Books	Lincolnwood, Ill., USA NTC Business Books

subject	notes
----------------	--------------

International business
enterprises Management

For example to insert a new publication in the database schema of DLAlert .with Public_id=1000 and Author='Giannis Alexakis' we have the following SQLJ code.

```
String NewPublic_id=1000;
String NewAuthor='Giannis Alexakis';
#sql {
    INSERT INTO alert.publications (PUBLIC_ID,AUTHOR)
    VALUES ( :NewPublic_id, :NewAuthor)
};
```

The actual SQLJ statement used for the transaction is.

```
#sql {
    INSERT INTO alert.publications
        (PUBLIC_ID, TITLE, SERIES, AUTHOR, PUBLISHER,
        SUBJECT, NOTES, PUB_YEAR, BOOKN, SERIALN)
    VALUES
        (
        :Publication_Id, :Title, :Series, :Author,
        :Publisher, :Subject, :Notes, :Year,
        :BookNumber, :SerialNumber )
};
```

The variables are assigned with the values to be inserted. Iterating through all the records we insert all of the new publications requested and parsed previously.

The translated or compiled Java code produced by the SQLJ translator can be loaded and executed inside the Oracle database. Java applications executed inside the RDBMS environment use the server-side internal JDBC for Oracle. As soon as we use this type of driver it is not necessary to explicitly declare a statement that establishes a JDBC connection with the database. The code executed inside an RDBMS is implicitly considered that references the same database.

In order to be able to call the method, that requests records from the JZKit API, parses them, and stores the bibliographical attributes of new publications, we have to

publish it as a Java Stored Procedure. The Java Stored Procedure declared (`ReadFromLibrary ()`) calls the Observer and returns the integer 1 on abnormal termination (for example due to network failure when a Z-association with the Gateway can not be established). In case of exception no records are inserted in the Oracle database.

7.6 Performance

The time needed to retrieve records from the Digital Library of TUC is mainly dependent to the network congestion between the Oracle database and the Gateway. It takes usually less than five minutes to retrieve about one thousand records from the Gateway since a single response contains 33 records at maximum. The time needed for parsing and the insertions is insignificant, as it is less than ten seconds for a thousand of records.

7.7 Important technical issues

We have the following important technical problems with the Digital Library of TUC that do not allow us deploy DLAlert in complete function yet:

- Most records that are inserted in the Digital Library of TUC have the date of acquisition field empty. The total number of records inside in the Digital Library is close to 60000 and the number of those with the date of acquisition filled is less than 6500. This means that almost the 90% of the records inserted in the database cannot be retrieved using this mechanism. This problem can be easily solved with the cooperation of the Library of TUC as long as we ensure that only future inserts/updates contain this essential bibliographical attribute. There is no need to change the data already in the database of the Library because we focus on new
-

assume this issue solved the (in any way) and propose scheduling for the actions of DLAlert.

- The Greek character set supported by the Digital Library is a non-standard custom
-

Chapter 8

Scheduling DLAlert

For the system to operate properly and on a regular basis we must schedule all the related modules and actions. For this purpose we can use the PL/SQL package DBMS_JOB [30, 33, 35, 40] which provides functionality for:

- Scheduling stored procedures to run unattended at some time in the future or upon certain intervals of time.
- Handling jobs that are broken for any reason (network or power failure, database error etc). These jobs are attempted to run 16 times if are not successfully executed.

We will not focus on the package, since scheduling the database is a rather complex administrator's task. We explain the sequence of actions to be executed, focusing on two simple scenarios. We discuss the two cases, of supporting or not different user categories according to their preferred frequency of notifications.

8.1 Simple scenario



Figure 8.1-1 Simple scenario sequence

For the case that we do not support different categories of users according to their preferred frequency of notifications we have the actions to be executed regularly (every day or week for example).

- First we have to collect new publication records from the Digital Library inserted during a certain interval of time. As the first step we call the module Observer.
- Synchronization of the CTXRULE indices is always necessary before filtering in order to have a consistent index with the base table of queries. For this purpose we have developed the PL/SQL package “indexing” (Section 4.4). This process can also be executed in the background during the first step, since the Observer is not an intensive CPU process.

- After synchronizing the indices we find matching profiles for every new publication (PL/SQL package “filtering”).
- Once the matching profiles are collected we can summarize all matched publications for every user and transmit e-mail messages via SMTP.
- As e-mail messages containing all relevant bibliographical attributes are constructed and delivered there is no need to maintain the already filtered documents. Unless we want to provide other functionalities among alerting services (for example information retrieval on the documents stored in the Oracle database) we can delete the publication records.

8.2 Supporting three types of desired notification frequencies

In order to support different notification frequencies we have three sets of actions as show in the above diagram. We categorize the actions according to their interval between two subsequent operations. “Every day” actions are executed every day regardless if this day is an end of week or month too. For example at the end of each month all three sets are executed sequentially.

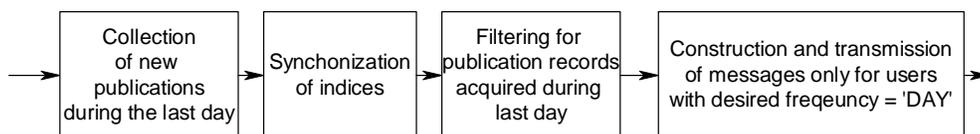


Figure 8.2-1 Actions executed every day

1) “Every day” actions.

- The first step is to collect new publication records from the Digital Library inserted during the last day. As the first step we always call the module Observer.
- Synchronization of indices is necessary in order for filtering to produce results consistent with the base table of queries.
- The next step next is to filter the publication records inserted during the last day.

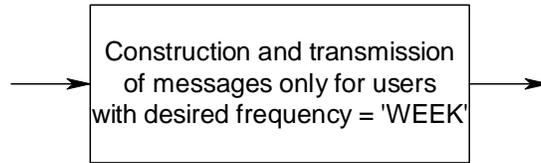


Figure 8.2-2 Actions executed once in a week

2) “Every week” actions.

- Since we have already inserted and filtered publications for every single day of the week, the only action that remains is to construct and send e-mail messages to users with ‘WEEK’ as the desired notification frequency.

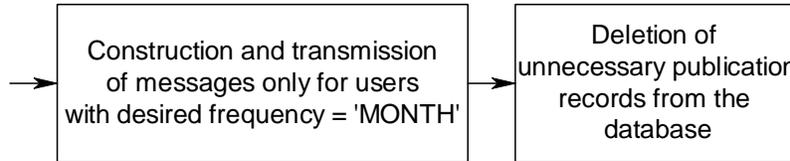


Figure 8.2-3 Actions executed once in a month

3) “Every month” actions.

- We have already inserted, filtered publications for every day up to the end of the month. Since we have already notified users of the first two categories (‘DAY’ and ‘WEEK’) the action that remains is to construct and send e-mail messages to users with ‘MONTH’ as the desired notification frequency.
- As e-mail messages containing all relevant bibliographical attributes for publications over the last month, are constructed and delivered, there is no need to maintain the already filtered documents. Optionally we can delete the unnecessary publication records from the Oracle database.

8.3 Conclusions

We have completed the presentation of the implementation and the development of DLAlert. We think that with minor configuration changes mainly on the Digital Library of TUC (Section 7.7) this system could easily be deployed to complete function and operate on regular basis. In last sections we presented two operating scenarios of DLAlert and, the corresponding actions to be scheduled in order to achieve the desired target. In the next chapter we propose future work on DLAlert.

Chapter 9

Concluding remarks

We think that an alerting service such as the one already developed, would be proved very helpful to the academic community of the Technical University of Crete. DLAlert should be enhanced with more functionalities like Greek support, integration of various sources and an even easier to use web interface. In addition DLAlert, a search engine that will support several information providers is being developed by the Intelligence Systems Laboratory. In the following section we purpose future work on DLAlert.

9.1 Future work on DLAlert

The following functionalities are proposed future enhancements on DLAlert. We think that if some of these are supported, DLAlert could be a popular alerting service as long as there is not any similar system developed in Greece at the time.

➤ **Greek support**

Greek support is a very important issue since most records in the Digital Library oM.098nk

(Synonym, and Broader, Narrower or Related Term). In order to support this functionality in Greek we should extract the main concepts found in the documents of the Digital Library and organize them hierarchically.

➤ **XML records classification**

Instead of using records containing the bibliographical attributes in plain text that represent incoming publications, we can represent publications as XML documents with sections defined as tags. For example consider the following example where we have a publication with Title: “The international business book” and Author: “Vincent Guy, John Mattock”. The corresponding XML document would be

```
<publication record>
  <title> The international business book </title>
  <author> Vincent Guy, John Mattock </author>
</publication record>
```

Oracle Text provides query operators for XML section searching like the operator *WITHIN*. We use the *WITHIN* operator to narrow a query down into document sections. For example to request documents with Title containing the word “business” we issue the *CTXRULE* query.

```
business WITHIN title
```

This approach has several advantages and disadvantages:

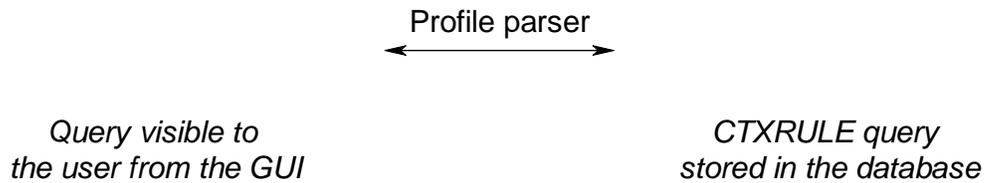
- Allows even more complex queries referencing sections that are not included in the current implementation of *DLAlert* (like “Anywhere” clauses). Using this approach we can easily request documents containing a keyword in any attribute or a custom set of attributes. We can even declare nested sections on records.
 - The filtering module will speed up in this case since we will need only one *CTXRULE* index for the text queries regardless the number of attributes supported. The time consumed for indexing will not be improved since it is mainly dependent on the total number of queries inserted / updated.
 - You cannot combine the *WITHIN* operator with the *ABOUT* operator, therefore we cannot request themes inside sections.
 - Requires a more complex parser for the profiles since queries referencing more than one attributes must be concatenated into a single *CTXRULE* query before inserted into the database. The operator *WITHIN* should not be visible to the user and the text query stored in the database should be re-parsed and broken into
-

simple *CTXRULE* statements before displayed on the GUI. For example to request documents with Title containing the word “business” and author containing the word “John” we issue the *CTXRULE* query.

Author	John
Title	business

➤ **Automatic word stemming expansion on queries**

Instead of expecting the user to enter the symbol \$ in order to request tokens with the same linguistic root as the requested term we can enhance the parser in order to automatically put the stemming symbol \$ before all queries. As we said previously this functionality cannot support the Greek language at the time, in case we use the supplied Oracle stemmer. For example suppose we request documents that contain the words “business” and “management”. DAlert can automatically include all the tokens with the same linguistic root as equivalent terms to the requested keywords.



Library of Technical University of Crete is represented by a single record inside the database (sample record on the following picture). Therefore DAlert cannot notify users on each number of the journal yet, but sends an e-mail message on a new subscription from the Library. Supporting specific journal requests on Profiles, is an essential feature supported by most popular Alerting Services on scholarly material (Section 2.1). The journals supported could be organized hierarchically according to their scientific area. Users should be notified regularly not only on a new journal subscription but also on each separate issue.



Figure 9.1-3 Sample record of a journal

➤ **Hyper-links in e-mail messages**

Providing all the bibliographical attributes of new publications on e-mail messages is an accurate way of notifying the user at the time. If we want to provide more information on new publications (like Table of Contents), it would be preferred to include hyper-links to web-pages containing all relevant data instead.

➤ **Notifications in various formats (plain text, HTML, XML)**

Providing notifications in various formats would be a useful feature. Some users may prefer shorter plain text e-mail messages. Also XML messages would be useful in case we send the notifications to another alerting service or application.

➤ **Using DIAS algorithms inside the database as Java stored procedures**

Functionality developed in the DIAS project could be integrated into the Oracle database, in case an implementation in Java that handles database records is available in the future. As we explained in Section 2.2.4 DIAS provides efficient algorithms for document filtering and profile matching. In this case the use of Oracle Text and the CTXRULE index would not be necessary. Java Stored Procedure technology enables us integrate almost everything that can be implemented in Java, as stored procedure inside the Oracle database.

➤ **Ranking of matching documents according to relevance**

Ranking of matching documents according to relevance is not supported for the CTXRULE index type in the current version of Oracle Text. Trying to support this feature would require enhancing of the filtering functionality available now.

➤ **Relevance feedback**

Relevance feedback on notifications means that the user can evaluate the relevance of the delivered documents so that the ranking results are improved in later filtering. This feature is also not supported at the time for the CTXRULE index type, and will require much development work to implement it. Java Stored Procedure technology will be most useful in case we try to develop functionality with high computational complexity like enhancing the filtering mechanism already available.

9.2 Conclusion

The main achievement of this dissertation is the development of a centralized alerting service for the Digital Library of the Technical University of Crete with the ability to integrate many information providers. As long as technical issues presented in 7.7 are solved, DLAlert can be scheduled to operate in regular basis. We hope that this dissertation will be a good starting point for further work on this application.

- [10] C.D. Manning and H. Schutze. *Foundations of Statistical Natural Language Processing*. The MIT Press, Cambridge, Massachusetts, 1999.
 - [11] *National Information Standards Organization* : <http://www.niso.org>
 - [12] *Z39.50 Standard Maintenance Agency*. <http://www.loc.gov/z3950/agency/>
 - [13] *MARC standards, Library of Congress Network Development and MARC standards Office*. <http://www.loc.gov/marc/>
 - [14] *Extensible Markup Language (XML) 1.0 (Second Edition)* W3C Recommendation 6 October 2000. Available at: <http://www.w3.org/TR/2000/REC-xml-20001006.pdf>
 - [15] *Knowledge Integration JZkit*. <http://developer.k-int.com/products/jzkit/>
 - [16] *Universal Bibliographic Control and International MARC Core Programme*: <http://www.ifla.org/VI/3/p1996-1/UNIMARC.htm>
 - [17] *UNIMARC Manual : Bibliographic Format 1994*: <http://www.ifla.org/VI/3/p1996-1/sec-uni.htm>
 - [18] *Z39.50 Text Part 9: Type-1 and Type-101 Queries*: <http://www.loc.gov/z3950/agency/markup/09.html>
 - [19] *Bib-1 Attribute Set*. <http://lcweb.loc.gov/z3950/agency/defs/bib1.html>
 - [20] *Registry of Z39.50 Object Identifiers*: <http://lcweb.loc.gov/z3950/agency/defs/oids.html>
 - [21] *Oracle Technology Network*: <http://otn.oracle.com/>
 - [22] *Oracle Corporation*: <http://www.oracle.com/>
 - [23] *Oracle Text Application Developer's Guide Release 9.2* Oracle Corporation. http://otn.oracle.com/docs/products/oracle9i/doc_library/release2/text.920/a96517.pdf
 - [24] *Oracle Text Reference Release 9.2*. Oracle Corp. http://otn.oracle.com/docs/products/oracle9i/doc_library/release2/text.920/a96518.pdf
 - [25] *Oracle Text Documentation* <http://otn.oracle.com/products/text/content.html>
 - [26] *Oracle Text Discussion Forum* <http://otn.oracle.com/forums/text.html>
 - [27] *Oracle Text Technical Overview (The CTXRULE Indextype)*: http://technet.oracle.com/products/text/x/Tech_Overviews/text_901.html
 - [28] *ISO 2788:1986 Documentation -- Guidelines for the establishment and development of monolingual thesauri*.
 - [29] *ANSI/NISO Z39.19 - 1993 Guidelines for the Construction, Format, and*
-

[30]

- [44] *JavaMail 1.3 Release*, Sun Microsystems, Inc.
Available at: <http://java.sun.com/products/javamail/>
- [45] *Core JavaScript Guide 1.5*. 2000, Netscape Communications Corp. :
<http://devedge.netscape.com/library/manuals/2000/javascript/1.5/guide/>
- [46] Marty Hall, *Core Servlets and JavaServer Pages*, Sun Microsystems Press/Prentice Hall. Available at <http://pdf.coreservlets.com/>
- [47] *JavaServer Pages Documentation*, Sun Microsystems. Available at:
<http://java.sun.com/products/jsp/docs.html>
- [48] *OracleJSP Support for JavaServer Pages Developer's Guide and Reference*, Release 1.1.3.1 Oracle Corporation. Available at:
<http://otn.oracle.com/docs/tech/java/oc4j/pdf/jsp1131.pdf>
- [49] Peter Koletzke, Paul Dorsey, Avrom Faderman. *Oracle9i JDeveloper Handbook*. December 2002 McGraw-Hill/Osborne Media.
- [50] *Java C C Home page*: <http://www.experimentalstuff.com/Technologies/JavaCC/>
- [51] *SQLJ Developer's Guide and Reference*. 2002 Oracle Corporation. Available at:
http://otn.oracle.com/docs/products/oracle9i/doc_library/901_doc/java.901/a90212.pdf

Related dissertations

- [52] Stratos Ydraios. "*A query and notification service based on mobile agents for rapid implementation of peer to peer applications*", 2003, Department of Electronic and Computer Engineering, Technical University of Crete.
- [53] Chistos Tryfonopoulos. "*Agent-Based Textual Information Dissemination: Data Models, Query Languages, Algorithms and Computational Complexity*", 2002, Department of Electronic and Computer Engineering, Technical University of Crete.
- [54] Theodoros Koutris. "*Textual information dissemination in distributed agent systems: Architectures and efficient filtering algorithms*", 2003, Department of Electronic and Computer Engineering, Technical University of Crete.
- [55] Sotiris Diplaris, Dimitris Pratsolis. "*Development of statistic lompustic models for the Greek language with stemming and part of speech functionality*", 2001, Department of Electronic and Computer Engineering, Technical University of Crete.
-