# The CYCLADES Collection Service

Leonardo Candela Donatella Castelli Pasquale Pagano
Istituto di Scienza e Tecnologie dell'Informazione "Alessandro Faedo"
Consiglio Nazionale delle Ricerche
Area della Ricerca CNR di Pisa
Via G. Moruzzi, 1 - 56124 PISA - Italy
{L.Candela|D.Castelli|P.Pagano}@isti.cnr.it

**Abstract**

This report introduces a digital library service, termed *Collection Service*, designed to support the dynamic creation of virtual collections of documents. Collections are created by specifying a set of descriptive criteria that express the information needs of a given community. The Collection Service provides two capabilities: on the one hand it constitutes a tool for dynamically structuring the information space according the needs of particular user/communities, on the other it supports the implementation of more efficient digital library services.

The paper exemplifies this service by showing how it has been instantiated in the CYCLADES digital library system and how it supplies support for efficient and effective query routing.

# Contents

## 1 Introduction

The building of a virtual library from distributed information resources was first made feasible by the Z39.50 community [2]. The Z39.50 protocol allows complying systems to search on multiple distributed databases and view the selected information sources as a single library. Although a Z39.50 Profile for Access to Digital Collections has been developed, the Z39.50 Protocol has been mainly used for bibliographic databases. A recent trend in digital libraries (DLs) is building by aggregating content by a set of different heterogeneous sources [10] that may grows along the time. The aim of these new generation DLs is to serve not only the information needs of those communities for which the component sources were initially set up, but also the needs of other multidisciplinary communities whose interests span across various information sources. One of the main problems encountered in designing these DLs is the implementation of an efficient and effective resource discovery. The heterogeneity of the content and the huge dimensions of the stored information render this problem hard to solve. The most common solution used in the past consists in structuring the whole information space into a number of established content classes, possibly organized hierarchically. Before formulating her/his query a user is asked to navigate in the hierarchy and to locate the class that best satisfies his/her needs. This organization is based on some fixed set of criteria – e. g. subject, date, location – that reflects both the typology of the underlying information sources and the needs of the expected user communities. This solutions fails for expandable DLs. Each new document added to the registered sources, or stored into the new sources must be explicitly indexed according to the terms of the established organization. This organization may over the time becomes obsolete and not capable to satisfy anymore the needs of the new communities of users.

This paper presents a different approach to this problem that works also for this new class of DLs. This approach has been designed for component-based DLs, i. e. DLs that are built as a federation of co-operating networked services. It has been currently experimented within the framework of the Cyclades (IST-2000-25456) EU funded project but it quite general and can be applied in many others component-based architectural frameworks. The key elements of our approach is a new type of service, the Collection Service (CS). This service is a dynamic content space mediator that mediates between the real organization of the content space, i. e. the set of registered content sources, and an organization into virtual sets of documents, termed *collections*, that are meaningful from the perspective of the DL user communities. Via collections users can collect together a set of resources logically correlated in order to satisfy an information need and refer to those as an information unit. The most important characteristics is that this set of resources is characterized via logical criteria and it is dynamic, e. g. if a new resource meet the collection definition criteria then it become automatically part of the set of collection's documents. The CS accepts requests for the creation of new collections, expressed in term of a set of criteria and, by exploiting the information about the underlying architectural configuration, dynamically generates collection descriptive metadata that are disseminated on request to the other services.

The rest of the report is structured as follow: the next section describes the Cyclades system. Section 3 presents the functionality of the CS in detail, while Section 4 introduces its logical architecture. Section 5 and 6 presents two key functions that are implemented by the CS service: the automatic archive content description acquisition and the archive selection. Sections 7 describes in detail a how the CS has been implemented in the Cyclades system. Section 8 presents related work and Section 9 concludes.

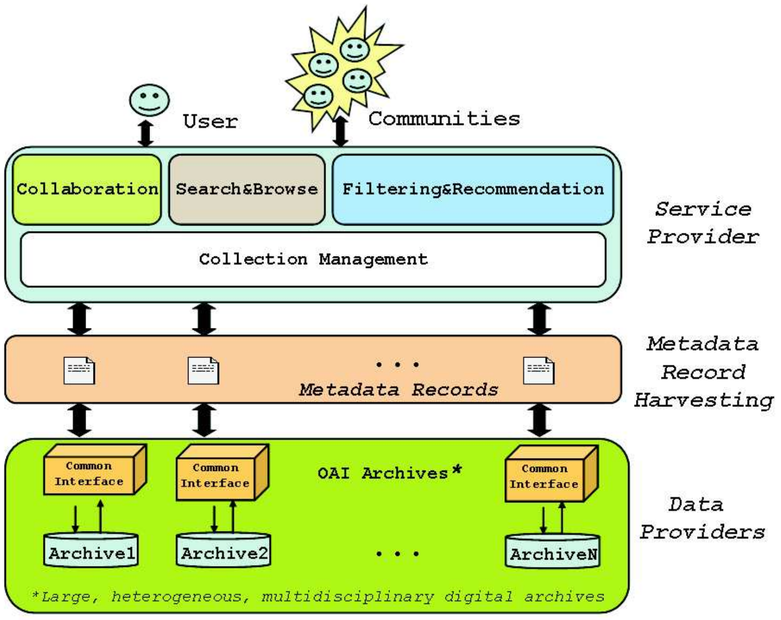


Figure 1: Logical view of Cyclades functionality.

## 2 Cyclades: a Personalized and Collaborative DL

Cyclades[1] has developed an open collaborative virtual archive service environment supporting both single scholars as well as scholarly communities in carrying out their work.

The objective of Cyclades is to provide an integrated environment for users and groups of users (communities) that want to use, in a highly personalized and flexible way, *open archives*, i. e. electronic archives of documents *compliant* with the Open Archive Initiative (OAI) [13]. The OAI develops and promotes interoperability standards that aim to facilitate the efficient dissemination of content. In particular, the OAI defines an easy-to-implement gathering protocol over HTTP, termed OAI-PMH [11], which give *data providers* (the individual archives) the possibility to make the documents' metadata in their archives externally available. This external availability of the metadata records then makes it possible for *service providers* to build higher levels of functionality. To date, there is a wide range of archives available in terms of its content, i. e. the family of OAI compliant archives is multidisciplinary in content. Under the above definition, Cyclades *is an OAI service provider* (see Figure 1) and provides functionality for *(i)* advanced search in *large, heterogeneous, multidisciplinary digital archives*; *(ii)* collaboration; *(iii)* filtering; *(iv)* recommendation; and *(v)* the management of records grouped into *collections*.

The Cyclades system architecture is depicted in Figure 2. It consists of a set of independent but interoperable services accessible via Web:

- The *Access Service* that is responsible for harvest-based information gathering, plus indexing and storage of gathered information in a local database;

- The *Collaborative Work Service* that supports collaboration between members of communities and project groups by providing functionality for creating shared working spaces referenc-

[1]`http://www.ercim.org/cyclades/`

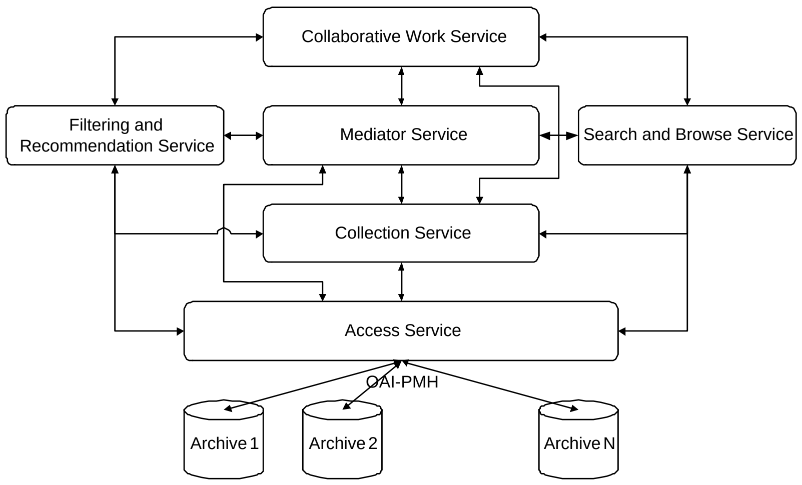

Collaborative Work Service
Filtering and Recommendation Service
Mediator Service
Search and Browse Service
Collection Service
Access Service
OAI-PMH
Archive 1
Archive 2
Archive N

Figure 2: CYCLADES Services Architecture.

ing users' own documents, collections, recommendations, related links, textual annotations, ratings, etc.;

- The *Filtering and Recommendation Service* that supports *(a)* information filtering on the basis of individual user profiles, and profiles of the working communities the user belongs to and *(b)* recommendations about new published articles within a working community;
- The *Search and Browse Service* that supports users in formulating queries, develops plans for their evaluation and provide an advanced multilevel browse facility completely integrated with the search facility;
- The *Mediator Service* that acts as a registry for the other services and provides security, i. e. it checks if a user is entitled to use the system, and ensures that the other services are only called after proper authentication.

The CS introduces a mechanism, the virtual collection, that allows users/communities to define their own information space, i. e. the set of documents they are interested in, via a set of characterization criteria. The most important feature of this mechanism is that the set of documents is dynamic, it reflects the dynamism of the CYCLADES information space builded as aggregation of a dynamic set of OAI-PMH compliant archives. Virtual collections make transparent the real, "by publisher" organization of the information space to the users. Each community, potentially each user, is enabled to tailor the whole CYCLADES information space on his needs and reorganizing it using the collection mechanisms.

By exploiting the reduction and readaptation of the information space to particular information needs, CYCLADES provides higher-quality result sets and a more faster search. This is in agreement to what is stated by Blair in [4] *"the best strategy for searching on a large system is to first reduce it to a small document collection"*.

## 3 Collection Service Functionality

CYCLADES allows users to follow the search strategy proposed in [4]. Here a two-stage search process is presented in order to improve the document retrieval on *large* collections. The first phase of this process consists of the *partitioning* of a large document collection into small collections (a partition), while the second phase consists of submitting the query representing the information needs to the right partition, i. e. the partition which is likely to contains the desired documents. The Collection Service supports the partitioning mechanism via the definition of *virtual collections*. A collection is usually defined as a statically identified set of documents. The CYCLADES collections are *virtual* as the system does not gather and store the documents belonging to a collection but it characterizes and identify them via a set of definition criteria. This means that CS collections are dynamic, i. e. they follow the dynamism of the underlying information space. If a new document meets the collection definition criteria then it is automatically included in the collection. The requests for the creation of new collections are submitted to the CS. These are formulated via a declarative collection definition language termed Membership Condition language that will be presented in Section 3.2.

Defined collection are stored by the CS and information about them is disseminated to the other services upon request. A collection is described by Collection Metadata, i. e. a set of data about the collection that comprises identification and managing information. The format and and the semantics of this metadata are described more in detail in Section 3.1.

The collection metadata are generated by a stepwise process that is composed by the following phases:

1. Via the CS GUI (Section 7.2) the user expresses his own information need using a definition language (Section 3.2). Note that this kind of information need is not a one-time request, i. e. is not intended for the identification of the single document the user is interested in, but it represents an expression of interest about a set of documents with certain characteristics where further to search in for a document;

2. The system processes the request of the user in order to *generate* the collection. During this phase detailed data about the collection (see Section 3.1) are derived by the system using a set of internal and automatic procedures. The most important procedure identifies the documents that belong to the collection. These documents are characterized by the set of characterization criteria that forms the Retrieval Condition (see Sections 3.1 and 6).

3. The collection is now ready to be *consumed* by other CYCLADES' services. The main functionality that CYCLADES implements over the collection mechanism is the two-stage search process. The *Search and Browse Service* allows, via its GUI, to select one or more collections where to search in. The *Filtering and Recommendation Service* is able to recommend a collection, i. e. an entire set of documents with a certain topic, to a user if it meets the user profile.

In what follows we will describe in more details the first two phases of the process described above.

## 3.1 Collection Metadata

Collection Metadata is the information that the system stores about a collection and disseminates upon request. It is composed by the following fields:

- *Identifier* - the unique identifier of the collection;

- *Name* - the name of the collection;

- *Description* - the textual description associated with the collection;

- *Membership Condition (MC)* - the condition that the creator has used to define the collection. It is maintained as a formal specification of the collection;

- *Retrieval Condition (RC)* - the condition that specifies how to retrieve, effectively and efficiently, the documents belonging to the collection;
- *Parent* - the identifier of the parent collection. It is used to maintain the hierarchical organization in the set of collections.

This is the minimal set of fields i required to manage collections. It contains identification information (Identifier, Name and Description), information on how to formally (MC) and operationally (RC) retrieve the content of the collection and information (Parent) about its position in the the hierarchical organization of the set of collections. This set of fields can be extended, with other kind of information - e. g. statistics about content, policies to regulate the access, and so on - in order to allow other service to have a more rich and detailed description of the collection. The richer this set of fields is the more accurate is the functionality that the other services can supply building over collections.

The main issue that the CS comes up against is the automatic derivation and generation of these metadata fields. A lot of them, e. g. Name, can be derived directly from the definition criteria expressed by the user, others are generated by the system, e. g. Identifier, whereas others require supplementary knowledge that the CS must either receive as input or acquire automatically. Section 5 and 6 discuss the latter case in more detail.

## 3.2 Membership Condition Language

The CS allows users to specify their own information needs via a declarative collection definition language termed Membership Condition Language. This language must be simple, expressive and quite powerful to capture any kind of information need arising from users. On the other hand, the definitions given in this language must be translated into a condition that all the information sources constituting the information space understand.

The syntax of the language that has been used in the prototype realization of the Cyclades CS is given below using the Backus-Naur Form (BNF):

```
query       ::= condition* [, (archiveList)]
condition   ::= ([weight,] field, predicate, value)
weight      ::= + | - | 1..1000
field       ::= [schemaName":"]attributeName
predicate   ::= cw | < | <= | >= | > | = | !=
archiveList ::= archiveName | archiveName, archiveList
```

This is an ALTAVISTA-style language where a query is a set of conditions, which are either optional, *mandatory* (+) or *prohibitive* (-). In addition, it allows for weighting of optional conditions (e. g. for relevance feedback). With respect to the structure of metadata records it assumes that they have a one-level structure and allows for the use of namespace (`schemaName`). The set of predicate supported is composed from the classical comparison operators ($<$, $<=$, $>=$, $>$, $=$ and $!=$) plus `cw` operator used to specify a condition on the content of a text field, e. g. `(description, cw, library)` stands for "the field `description` contains the term `library`".

This language has pros and cons:

- it is quite simple and intuitive as it is similar to others, well known query languages;
- it is quite general, the assumption about the one-level metadata record structure can be simply removed using the attribute name path instead of the attribute name;
- it is not enough expressive as others query languages are, e. g. SQL. We are currently working at the evaluation of the *right grade of expressive power* required in order to define collections.

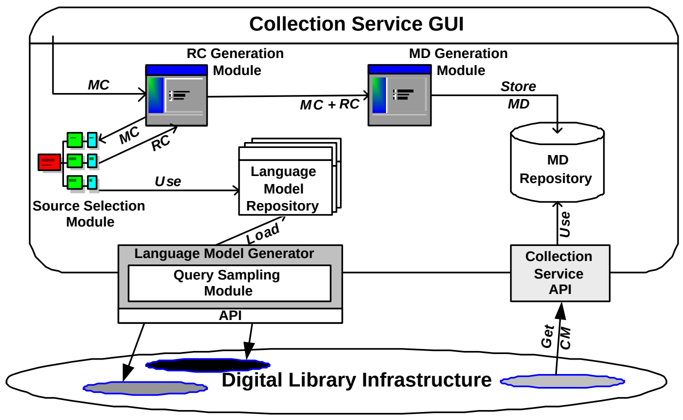


Figure 3: The Cyclades Collection Service Logical Architecture.

## 4 Collection Service Architecture

Figure 3 shows the logical architecture of the Cyclades CS. This picture shows how the initial user description of the collection, i. e. the membership condition *MC*, is manipulated in order to produce the collection metadata *MD* that are stored into the system and disseminated upon request via the CS API, in accordance with the process presented in Section 3.

The CS contains a module, the *Retrieval Condition Generation*, that is responsible for the generation of the retrieval condition *RC*, i. e. the condition that is used in order to find the documents belonging to the collection. The RC consists of the membership condition plus a set of automatically selected archives[2] that are relevant to the conditions specified via the membership condition.

In order to identify this set of archives the *Retrieval Condition Generation* module uses the *Source Selection Module*. This module is responsible for the solving the source selection problem, i. e. the selection of the subset of information sources relevant to a given query among the set of accessible sources (see Section 6). Cyclades can be considered a distributed search environment. As previously stated the *Access Service* is responsible for harvest-based information gathering, plus indexing and storage of gathered information in a local database. From the design choice of Cyclades the *Access Service* does not have a global index for all the archives, instead it has an index for each archive. If a query is over the whole Cyclades information space, i. e. if none of the archives have been specified, the *Access Service* dispatches the query to all the archives and then merges the results to produce the result of the query.

In order to choose the *right* information sources, the CS must *know* them, i. e. it must have an appropriate knowledge of the content of each information source. How to best represent an information source content is an open problem. The current approach to this problem is based on the use of lists of terms with their frequency or term weight information also known as *language model*. This approach will be presented in more detail in Section 5 where a technique used for acquiring the language model from a set of non cooperating information sources is reported.

Finally, it is important to note that even though some of the logical modules must interact directly with the DL infrastructure and, therefore, part of their design is strictly dependent on that environment, the CS can be considered infrastructure independent because only a minimal API set has to be realized.

[2]In the follow we will use the terms archive and information source as synonyms.

## 5 Language Model and Query-Based Sampling

Cyclades elaborates a query using the classical approach adopted in a distributed search environment: (a) it reformulates the query for each information source – if necessary –, (b) it sends the appropriate query to each archive and (c) it merges the results returned. In order to improve the efficacy of this process we have introduced a pre-phase termed *source selection* that aims to select the relevant information sources for the query.

The purpose of translating the information needs expressed by the user via the collection's Membership Condition into the collection's Retrieval Condition is twofold: (i) on the one hand it is necessary to rewrite the MC into the query language supported by the information source in order to allow it to reply; (ii) on the other hand it is important to optimize, to module the information needs expressed by the MC considering the actual state of the underlying information space. The latter aspect is related with the source selection: the CS generates a Retrieval Condition that shows to the Cyclades *Access Service* the information source to be queried in order to supply the information need expressed by the collection.

In order to select the relevant information sources for a query, the CS must have a description of their content. From our point of view an information source is a set of documents. The issue of how to best describe this set is an open problem. The most widely used approach in the literature consists in using a *language model*, i. e. a list of terms with their term frequency or term weight information. As it will be clarified in the next section, this knowledge is sufficient for the source selection technique that we have adopted.

In the Cyclades framework the information sources do not supply their own language model as usually happens in a federated search environment. Each information source exposes his own content, i. e. the set of records it maintains, via the OAI-PMH and the *Access Service* gathers this content and stores it into a local database supplying to other services *only* query functionality. By exploiting this characteristics the CS is able to acquire the language model using the *query-based sampling* technique.

The query-based sampling technique has been proposed by Callan and Connell [6] for acquiring accurate resource description[3] in a context where information sources are text databases. This technique does not require the cooperation of source providers, nor does it require that source providers use a particular search engine or presentation technique. Resource descriptions are created by running queries and examining the documents returned. At the end of this process a sample of the records of the information source that represent its content is acquired. This set is called resource description and using it the language model of the archive can be derived.

```
1:  query = generateInitialTrainingQuery();
2:  resultSet = run(query);
3:  if(|resultSet| < $L_{tr}$){
4:      go to 1;
5:  }else{
6:      updateResourceDescription(resultSet);
7:      if(NOT stoppingCriteria()){
8:          query = generateTrainingQuery();
9:          resultSet = run(query);
10:         go to 6;
11:     }
12: }
```

Figure 4: Sampling Algorithm.

[3]Resource description is a kind of knowledge about the content of an information source.

Figure 4 shows the query-based sampling algorithm that we have extended for databases with multiple text attributes – e.g. bibliographic records – (similar to [16]). This algorithm uses the functions explained below:

**generateInitialTrainingQuery()** generates the *start* training query. In order to generate a query we need: (a) a set of words among which randomly choose the ones to build the condition and (b) a set of attribute among which randomly choose the ones to build the condition. For each selected attribute we randomly select 1 to $max_t$ distinct terms and for each pair we choose an operator to relate attribute and term into the condition.

This function, like the generateTrainingQuery(), is dependent from the information source query language and from other parameters. Here we assume that each query language supports at least conditions on single attribute of a bibliographic record. All the other aspects are configurable.

In the Cyclades CS prototype we have taken the following design choices: (a) the words belongs to the set of terms that characterize the second and the third level of Dewey Decimal Classification [1] , (b) the attributes that we have used belongs to the Dublin Core[4] fields, (c) $max_t = 4$ and (d) the operator used is always the `cw` operator – the query language supported by the Access Service is similar to the one presented in Section 3.2.

**updateResourceDescription()** updates the set of records that represents the resource description. Note that a query must return at least $L_{tr}$ records before the records collected (the top $L_{tr}$) can be added to the resource description record set. This minimum result size is required because query returning small results do not capture source content well.

In our prototype we have use $L_{tr}$ equals to 4 as proposed in [16], this is just another configuration aspect.

**stoppingCriteria()** evaluates if the stopping criteria was reached. For our best knowledge no one has proposed significantly stopping criteria.

Callan and Connell [6] experiments have been conducted stopping the sampling after examining 500 documents, a stopping criteria chosen empirically observing that augmenting the number of documents examined the language model does not improve significantly.

In our prototype implementation the stopping criteria is reached when the system runs 10 queries, each ones returns at least $L_{tr}$ records without resource description records set changes. This is an aspect that we plan to further investigate in the future.

**generateTrainingQuery()** generates the *next* training query. Training queries are generated as follow:

1. randomly select a record $R$ from resource description record set;
2. randomly select a set of attribute of $R$ to use in training query;
3. for each attribute to be included in the training query, construct a predicate on it by randomly selecting 1 to $max_t$ distinct terms (stopwords are discarded) from the corresponding attribute value and using the `cw` operator.

In order to investigate the accuracy of the learned resource description acquired via the sampling technique, we have conducted some experiments whose results are reported below.

[4]`http://dublincore.org/`

We have considered two bibliographic information source, *Archive 1*, quite small and homogeneous[5] information source, and *Archive 2*, larger and heterogeneous[6] information source.

The experimental method was based on comparing the learned resource description of an information source with the real resource description for that information source. Resource description can be represented using two information, a vocabulary V of the set of terms appearing in the information source records and a frequency information for each vocabulary term. This frequency, also called *document frequency* (*df*), represents the number of documents containing the terms. In accordance with [6] we have used two metrics to evaluate the quality of the resource description acquired by sampling, the *ctf ratio* (CTF) to measure the correspondence between the learned ($V'$) and the real ($V$) vocabulary and the *Spearman Rank Correlation Coefficient* (SRCC) to measure the correspondence between the learned and the real frequency information.

$$\mathrm{CTF} = \frac{\displaystyle\sum_{i \in V'} ctf_i}{\displaystyle\sum_{i \in V} ctf_i} \tag{1}$$

$$\mathrm{SRCC} = 1 - \frac{6}{n^3 - n} \sum {d_i}^2 \tag{2}$$

This metrics are calculated using formulas 1 and 2 where:

- $ctf_i$ is the number of times terms $i$ occurs in the resource description of an information source,
- $d_i$ is the rank difference of common term $i$ where term rankings are produced by learned and actual *df* values,
- $n$ is the number of terms.

Five trials were conducted for each information source and for each trial the resource description of 500 records has been acquired. The results reported here are the average of the results returned by the trials.

Figures 5 and 6 shows respectively the CTF and the SRCC metrics calculated by field and by record for *Archive 1*, varying the number of records considered building the resource description acquired by sampling, while Figures 7 and 8 shows the same metrics calculated for *Archive 2*.

By observing the CTF graphics we note that the language model acquired for the first archive is better then the one acquired for the second one. Moreover we can note that the language model acquired for a field has different characteristics than the one acquired for others field. The reasons for this behavior are twofold: *Archive 2* contains more records and is more heterogeneous than *Archive 1* and some fields, e. g. `creator`, are more heterogeneous than others, e. g. `date`.

By observing the SRCC graphics we can note that the *quality* of the language model acquired via sampling is high, considering the record as a plain text we can found values greater that 80% (see RECORD line).

During our experiments we have not considered the stopping criteria described earlier. The quality of the results obtained – we have not the same regularity – has suggested us to consider a different stopping criteria in sampling procedure than that proposed by Callan.

A more complete and detailed study of this technique is necessary and we plan to do this in the near future. The positive and interesting aspect is that the source selection works well with the sample acquired using our algorithm as showed in the next section.

[5]1616 records, 13576 unique terms after stopwords removing, papers about computer science published by the same authority.

[6]16721 records, 79047 unique terms after stopwords removing, papers published by different authorities.

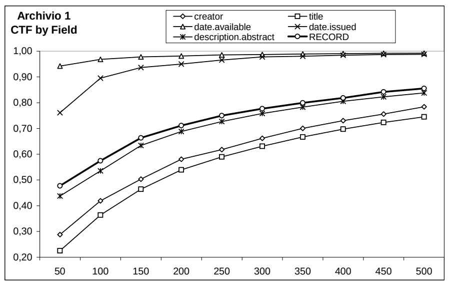


Figure 5: Archive 1: CTF.

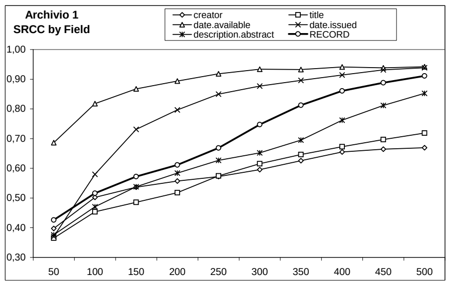


Figure 6: Archive 1: SRCC.

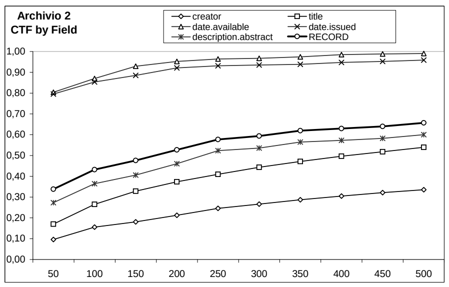


Figure 7: Archive 2: CTF.

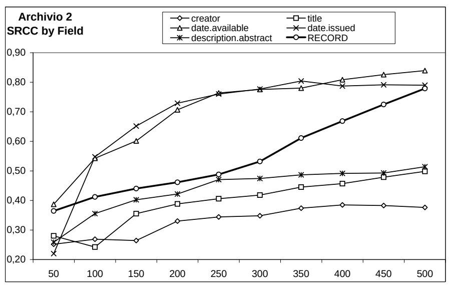


Figure 8: Archive 2: SRCC.

## 6 Source Selection Technique

As already stated, source selection is the problem of selecting from a large set of accessible information sources the ones relevant to a given query. In our case is the query is the MC, i. e. the collection characterization criteria, while the selected information sources are used in the generation of the Retrieval Condition in order to allow a faster discovery of the documents belonging to the collection.

The source selection problem can be formally defined as follow:

*Let $\overline{IS} = \{IS_1, IS_2, \ldots, IS_N\}$ be a set of Information Sources. Let $q$ be a query. Compute $E \subseteq \overline{IS}$ such that $\forall\, F \subseteq \overline{IS}$ Goodness(q,E) $\geq$ Goodness(q,F).*

*Goodness* is a function on the results returned by a set of IS $E$ against a query $q$ defined as follow:

$$Goodness(q, E) = \sum_{IS_i \in E} s_i$$

where $s_i$ is the result size returned by $IS_i$ for query $q$.

In order to obtain the maximum *Goodness* value for a query it is sufficient to rank the information sources estimating the result size returned by each one. The weighting scheme that we propose has been obtained extending the CORI scheme [5] in order to manage bibliographic records instead of text documents and to consider a richer query language than a keyword-based ones.

In accordance with the Membership Condition Language reported in 3.2 we consider a keyword-fielded-based query model, this mean that a query $q$ is defined as a list of condition $(w_i, a_i, o_i, v_i)$ where:

- $w_i$ is the *weight* of this condition. + mean that the condition must be fulfilled, - that the condition must not be fulfill (the boolean NOT);
- $a_i$ is the field of the bibliographic record involved in the condition;
- $o_i$ is the operator to use, e. g. `<=`, `=`, `cw`, etc.;
- $v_i$ is the keyword.

For example, to retrieve all the records having author "Castelli" and subject "Cyclades" we use the following query:

```
(+,author,cw,''Castelli'')(+,subject,cw,''Cyclades'')
```

The technique exploits the discriminatory powers of different conditions to increase the accuracy of archive selection. This is done by summarizing the content of the information source $IS$ via the language model $LM$. As stated in Section 5 the language model consists of a list of terms with their term frequency and it is acquired by sampling. Using it the CS is able to calculate the *document frequencies* (denoted by $df_{i,j}$) defined as the expected number of records in $IS_i$ that match against the condition $c_j$ plus other statistical values described in the follow.

Formally, the *Goodness* score $G(q, IS_i)$ for IS $IS_i$ and query $q$ is defined as follow:

$$G(IS_i, q) = \begin{cases} 0 & \text{if } \exists k \in [1..|q|] \mid w_k \in \{+, -\} \wedge p(c_k|IS_i) = 0 \\ \dfrac{\sum_{k=1}^{|q|} p(c_k|IS_i)}{|q|} & \text{otherwise} \end{cases} \tag{3}$$

where the "belief" $p(c_k|IS_i)$ in $IS_i$ for condition $c_k$ is defined as

$$p(c_k|IS_i) = \begin{cases} T_{i,k} \cdot I_k \cdot w_k & \text{if } w_k \in [1..1000] \\ T_{i,k} \cdot I_k & \text{if } w_k = \text{“}+\text{” or } w_k = \text{“}-\text{”} \end{cases} \tag{4}$$

$$T_{i,k} = \frac{df_{i,k}}{df_{i,k} + 50 + 150 \cdot \dfrac{cw_{i,k}}{\overline{cw_k}}} \tag{5}$$

$$I_k = \frac{\log\left(\frac{|D|+0.5}{cf_k}\right)}{\log\left(|D| + 1.0\right)} \tag{6}$$

where:

| | |
|---|---|
| $df_{i,k}$ | is the expected number (estimated via $LM_i$) of documents in $IS_i$ satisfying $c_k$, |
| $cw_{i,k}$ | is the number of terms in attribute $a_k$ in $LM_i$, |
| $\overline{cw_k}$ | is the mean $cw$ of the ISs being ranked, |
| $cf_k$ | is the number of ISs that satisfying $c_k$, |
| $\vert D\vert$ | is the number of the ISs being ranked. |

Note that the accuracy of the automatic source selection using this technique is promising, i. e. the RC that is generated approximates very well the MC. This is demonstrated by a set of experiments the we carried out on a Cyclades configuration that was working on 62 OAI compliant archives. In particular, in these experiments we generated randomly 200 collections using Dublin Core fields. The collections generated are of two kinds: 100 collections (T1) are generated using a combination of conditions on `description` and `title` fields, 100 collections (T2) are generated using a combination of conditions on all fields of the Dublin Core schema.

Table 1 shows the results of some preliminary tests on the quality of the source selection. *Precision* is defined as $gcd/(gcd + bcd)$ and *recall* is defined as $gcd/(gcd + gncd)$. In order to calculate this values we submit the $MC_i$ query to all the archives in Cyclades obtaining a set

| | | Precision | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | 0.00 – 0.10 | 0.11 – 0.20 | 0.21 – 0.30 | 0.31 – 0.40 | 0.41 – 0.50 | 0.51 – 0.60 | 0.61 – 0.70 | 0.71 – 0.80 | 0.81 – 0.90 | 0.91 – 1.00 | |
| | 0.00 – 0.10 | 0.33% | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 8% | 8.33% |
| | 0.11 – 0.20 | 0 | 0 | 0 | 0 | 0 | 0.16% | 0 | 0 | 0 | 5.83% | 6% |
| R | 0.21 – 0.30 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 5.83% | 5.83% |
| e | 0.31 – 0.40 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 7.5% | 7.5% |
| c | 0.41 – 0.50 | 0 | 0 | 0 | 0 | 0 | 0.16% | 0 | 0 | 0.16% | 12.16% | 12.5% |
| a | 0.51 – 0.60 | 0 | 0 | 0 | 0 | 0 | 0.16% | 0 | 0 | 0 | 2.5% | 2.66% |
| l | 0.61 – 0.70 | 0 | 0 | 0 | 0 | 0 | 0 | 0.16% | 0 | 0 | 8.66% | 8.83% |
| l | 0.71 – 0.80 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0.5% | 0.33% | 8.83% | 9.66% |
| | 0.81 – 0.90 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1.33% | 9.83% | 11.16% |
| | 0.91 – 1.00 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 27.5% | 27.5% |
| | | 0.33% | 0 | 0 | 0 | 0 | 0.5% | 0.16% | 0.5% | 1.83% | 96.66% | |

Table 1: Source selection: precision and recall.

| | T1 | T2 | Average |
|---|---|---|---|
| MC | 162874 ms | 186909 ms | 174892 ms |
| RC | 48469 ms | 52253 ms | 50361 ms |
| Improvement in ms | 114405 ms | 134655 ms | 124530 ms |
| Improvement in % | 70.24% | 72.04% | 71.20% |

Table 2: Source selection: average response time.

of records $Ret_{MC_i}$, then we submit the $RC_i$ to the archive specified in it obtaining a new set of records $Ret_{RC_i}$. *gcd* is the number of documents that belong to the collection that are correctly retrieved, i. e. $|Ret_{MC_i} \cap Ret_{RC_i}|$, *bcd* is the number of documents that do not belong to the collection that are erroneously retrieved (also called *false positives*), i. e. $|Ret_{RC_i} \setminus Ret_{MC_i}|$, and *gncd* is the number of documents belonging to the collection that are not retrieved (also called *false negatives*), i. e. $|Ret_{MC_i} \setminus Ret_{RC_i}|$. The calculated pairs of this values have been partitioned into precision/recall intervals and reported into the table as percentage. Moreover the right most column and the bottom row shown the total amount w. r. t. a row and a column, respectively. For instance, we have that 27.5% of the test cases have a recall level in [0.91,1], while the 96,66% of the test cases have a precision level in [0.91,1].

The RC effectively improves the performance. Table 2 shows a measure of this improvement. In particular, it compares the query response times obtained by retrieving the set of documents matching the $MC_i$ with those obtained using the $RC_i$.

Table 1 and Table 2 shows that there is an high improvement in response time with little loss in the set of records retrieved after automatic source selection.

## 7 Implementation

One of our main goal in designing and implementing the Collection Service has been reusability and adaptability to different contexts. This section describes the choices that have been taken to achieve this goal as far as the implementation is concerned.

First, we have chosen to use Java as development tool in order to realize a fully portable service. Moreover, service functionality has been made accessible either via GUI using a Web browser – (see

Section 7.2), and via API using the XML-RPC protocol (see next section for a detailed description). Finally, we have used XML to represent collection metadata in order to enhance data portability.

Another interesting aspect is the configurability of the system. Many characteristics of the service are easily modifiable, e. g. some aspect of the Membership Condition Language, like the set of the attribute or the set of predicate supported, some parameters of the query based sampling, like the set of words to use in query generation or the number of documents to examine, etc.

## 7.1 The Collection Service API

The Collection Service provides an API to other services in order to allow them to easily access its own functionality. This section supplyies a full description of this API reporting the signature, a description of the method and of its parameters, and the set of exception raised for each method supported. Moreover in Section 7.1.2 the results of a set of tests conducted on this API are reported.

API methods can be called using the inter-service communication protocol XML-RPC[7]. This is a Cyclades project choice justified mainly by the characteristics of simplicity and wide support that this protocol has.

Invoking the Collection Service's methods such default exception can be thrown:

| | | |
|---|---|---|
| 10000 | no such method | Method invoked is not defined. |
| 10001 | bad number of parameters | Method invoked with a wrong parameters number. |
| 10002 | bad parameter type | Method invoked with a wrong parameter type. |
| 10010 | internal error | An *undefined* exception exists. |

- **Method:** addCollection
  **Signature:** collectionId addCollection()
  **Description:** this method creates a new collection identifier which can be assigned to a collection which will be created soon.
  **Parameters:**

| | | | |
|---|---|---|---|
| Output: | collectionId | integer | the identifier of the new collection that will be created. |

- **Method:** initializeCollection
  **Signature:** collectionId initializeCollection(collectionId, collectionName, collectionDescription, membershipCondition, userId)
  **Description:** this method creates a collection, whose parent collection is the *Cyclades* collection, if the membership condition is legal.
  **Parameters:**

| | | | |
|---|---|---|---|
| Input: | collectionId | string | the identifier of the new collection. |
| | collectionName | string | the printable name of the collection (max 50 chars). |
| | collectionDescription | string | textual description of the collection. |
| | membershipCondition | string | the condition to be verified by all the members of the collection coded in XML (see 7.1.1). |
| | userId | string | the identifier of the user which sends the request. |
| Output: | collectionId | string | the identifier of the collection that has been initialized. |

  **Exception:**

[7] `http://www.xmlrpc.com/`

| | | |
|---|---|---|
| 10003 | missing or null parameter value | if `collectionName` or `collectionDescription` are “”. |
| 14112 | User doesn’t exists | |
| 10200 | no permission | operation not allowed, user isn’t enable to do it |
| 14113 | Identifier is not valid | `collectionId` is not valid. |
| 14107 | Bad XML file | error parsing `membershipCondition`. |
| 14115 | Out of bounds | if `collectionName` is out of bounds. |

- **Method:** initializeCollection
  **Signature:** collectionId initializeCollection(collectionId, collectionName, collectionDescription, membershipCondition, userId, parentCollection)
  **Description:** this method creates a collection whose parent collection is parentCollection, if the membership condition is legal.
  **Parameters:**

| | | | |
|---|---|---|---|
| Input: | collectionId | string | the identifier of the new collection. |
| | collectionName | string | the printable name of the collection (max 50 chars). |
| | collectionDescription | string | textual description of the collection. |
| | membershipCondition | string | the condition to be verified by all the members of the collection coded in XML (see 7.1.1). |
| | userId | string | the identifier of the user which sends the request. |
| | parentCollection | string | the identifier of the parent collection in the collection hierarchy. |

Output: collectionId string the identifier of the collection that has been initialized.
**Exception:**

| | | |
|---|---|---|
| 10003 | missing or null parameter value | if `collectionName` or `collectionDescription` are “”. |
| 14112 | User does not exists | |
| 10200 | no permission | operation not allowed, user is not enable to do it |
| 14113 | Identifier is not valid | `collectionId` is not valid. |
| 14105 | Collection does not exists | `parentCollection` does not exists. |
| 14107 | Bad XML file | error parsing `membershipCondition`. |
| 14115 | Out of bounds | if `collectionName` is out of bounds. |

- **Method:** deleteCollection
  **Signature:** void deleteCollection(collectionId, userId)
  **Description:** this method removes a collection from the set of existing collections if: a) the user is authorized to do it and b) the specified collection exists.
  **Parameters:**

| | | | |
|---|---|---|---|
| Input: | collectionId | string | the identifier of the new collection. |
| | userId | string | the identifier of the user which sends the request. |

**Exception:**

| | | |
|---|---|---|
| 10200 | no permission | operation not allowed, user is not enable to do it |
| 14105 | Collection does not exists | `collectionId` does not exists. |

- **Method:** listCollections
  **Signature:** (collectionId, collectionName, collectionDescription, parentCollection)* listCollections(userId)
  **Description:** this method returns the list of existing collections whose owner is userId.
  **Parameters:**
  Input: userId string the identifier of the user who sends the request
  Output: a list of (collectionId, collectionName, collectionDescription, parentCollection) where:

| | | |
|---|---|---|
| collectionId | string | the identifier of the collection. |
| collectionName | string | the name of the collection. |
| collectionDescription | string | the description of the collection. |
| parentCollection | string | the identifier of the parent. collection . |

**Exception:**

| | |
|---|---|
| 14112 | User does not exists |

- **Method:** listCollections
  **Signature:** (collectionId, collectionName, collectionDescription, parentCollection)* listCollections()
  **Description:** this method returns the list of existing collections.
  **Parameters:**
  Output: a list of (collectionId, collectionName, collectionDescription, parentCollection) where:

| | | |
|---|---|---|
| collectionId | string | the identifier of the collection. |
| collectionName | string | the name of the collection. |
| collectionDescription | string | the description of the collection. |
| parentCollection | string | the identifier of the parent collection. |

- **Method:** editCollection
  **Signature:** void editCollection(collectionMetadata,userId)
  **Description:** Update collection metadata description.
  **Parameters:**

| | | | |
|---|---|---|---|
| Input: | collectionMetadata | string | new collection metadata coded in XML (see 7.1.1). |
| | userId | string | the identifier of the user who sends the request. |

**Exception:**

| | | |
|---|---|---|
| 14107 | Bad XML file | error parsing `collectionMetadata`. |
| 10200 | no permission | operation not allowed, user isn't enable to do it |
| 14105 | Collection doesn't exists | |

- **Method:** getCollectionMetadata
  **Signature:** (collectionId, collectionMetadata)* getCollectionMetadata(collectionIds*)
  **Description:** for each specified collection identifier, this method returns the corresponding descriptive metadata.
  **Parameters:**
  Input: collectionIds string* a list of collection identifiers.
  Output: A list of pairs (collectionId,collectionMetadata) where:

| | | |
|---|---|---|
| collectionId | string | the identifier of the collection . |
| collectionMetadata | string | the collection metadata coded in XML (see 7.1.1). |

- **Method:** getPersonalCollections
  **Signature:** (collectionId, collectionName, collectionDescription, parentCollection)* getPersonalCollections(userId)
  **Description:** this method returns the list of personal set of collections for user `userId`.
  **Parameters:**
  Input: userId string the identifier of the user who sends the request
  Output: a list of (collectionId, collectionName, collectionDescription, parentCollection) where:

| | | |
|---|---|---|
| collectionId | string | the identifier of the collection. |
| collectionName | string | the name of the collection. |
| collectionDescription | string | the description of the collection. |
| parentCollection | string | the identifier of the parent. collection . |

**Exception:**

| | |
|---|---|
| 14112 | User doesn't exists |

- **Method:** deleteUser
  **Signature:** void deleteUser(userId)
  **Description:** Notify the Collection Service that user `userId` was removed.
  **Parameters:**
  Input: userId string the identifier of the user.
  **Exception:**
  14112 User does not exists

- **Method:** deleteArchive
  **Signature:** void deleteArchive(archiveId)
  **Description:** Notify the Collection Service that archive `archiveId` was removed.
  **Parameters:**
  Input: archiveId string the identifier of the archive.

### 7.1.1 XML objects: XML schemas

As stated previously we have used XML to represent collection metadata in order to enhance data portability. Moreover the Membership Condition itself is described via an XML file, the schema can be used to validate the syntax of this file. In this section we report the XML Schema describing formally this information.

`Collection Metadata` **Schema**

```
<?xml version="1.0" encoding="UTF-8"?> <xs:schema
        targetNamespace="http://project.iei.pi.cnr.it:8080/CollectionService"
        xmlns:query="http://project.iei.pi.cnr.it:8080/CollectionService"
        xmlns:xs="http://www.w3.org/2001/XMLSchema"
        xmlns:cs="http://project.iei.pi.cnr.it:8080/CollectionService"
        elementFormDefault="unqualified" attributeFormDefault="unqualified">
  <xs:include schemaLocation="query.xsd"/>
  <xs:include schemaLocation="membership.xsd"/>
  <xs:element name="CollectionMetadata" type="cs:collectionMetadataType">
  </xs:element>
  <xs:complexType name="collectionMetadataType">
    <xs:sequence>
      <xs:element name="Id" type="xs:string"/>
      <xs:element name="Name" type="xs:string"/>
      <xs:element name="Description" type="xs:string"/>
      <xs:element name="OwnerId" type="xs:string"/>
      <xs:element name="ParentCollection" type="xs:string"/>
      <xs:element name="MembershipCondition" type="cs:MCType"/>
      <xs:element name="FilteringCondition" type="query:FCType"/>
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="FCType">
    <xs:sequence>
      <xs:element name="query">
        <xs:complexType>
```

```
          <xs:sequence>
            <xs:element name="collection-query" type="query:collection-queryType"
                minOccurs="0" maxOccurs="unbounded"/>
          </xs:sequence>
          <xs:attribute name="schema" type="xs:string" use="required"/>
        </xs:complexType>
      </xs:element>
    </xs:sequence>
  </xs:complexType>
</xs:schema>
```

`Membership Condition` **Schema**

```
<?xml version="1.0" encoding="UTF-8"?> <xs:schema
        targetNamespace="http://project.iei.pi.cnr.it:8080/CollectionService"
        xmlns:cs="http://project.iei.pi.cnr.it:8080/CollectionService"
        xmlns:xs="http://www.w3.org/2001/XMLSchema"
        elementFormDefault="unqualified" attributeFormDefault="unqualified">
  <xs:element name="MembershipCondition" type="cs:MCType"/>
  <xs:complexType name="MCType">
    <xs:sequence>
      <xs:element name="metadataFormat" type="xs:string"/>
      <xs:element name="condition" type="cs:conditionType" maxOccurs="unbounded"/>
      <xs:element name="archive" type="xs:string" maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="conditionType">
    <xs:attribute name="weight" type="xs:string"/>
    <xs:attribute name="field" type="xs:string" use="required" />
    <xs:attribute name="predicate" type="xs:string" use="required" />
    <xs:attribute name="value" type="xs:string" use="required" />
  </xs:complexType>
</xs:schema>
```

`Query` **Schema**

```
<?xml version="1.0" encoding="UTF-8"?> <xs:schema
        targetNamespace="http://project.iei.pi.cnr.it:8080/CollectionService"
        xmlns:xs="http://www.w3.org/2001/XMLSchema"
        xmlns:query="http://project.iei.pi.cnr.it:8080/CollectionService"
        elementFormDefault="unqualified" attributeFormDefault="unqualified">
  <xs:complexType name="archiveType">
    <xs:attribute name="id" type="xs:string" use="required"/>
  </xs:complexType>
  <xs:complexType name="collection-queryType">
    <xs:sequence>
      <xs:element name="condition" type="query:queryConditionType"
          minOccurs="0" maxOccurs="unbounded"/>
      <xs:element name="archive" type="query:archiveType"
          minOccurs="0" maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="queryConditionType">
    <xs:sequence>
      <xs:element name="field-condition" type="query:field-conditionType"
          maxOccurs="unbounded"/>
    </xs:sequence>
    <xs:attribute name="weight" type="xs:string"/>
    <xs:attribute name="field" type="xs:string" use="required"/>
  </xs:complexType>
  <xs:complexType name="field-conditionType">
    <xs:attribute name="subfield" type="xs:string"/>
    <xs:attribute name="predicate" type="xs:string" use="required"/>
    <xs:attribute name="value" type="xs:string" use="required"/>
  </xs:complexType>
  <xs:element name="query">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="collection-query" type="query:collection-queryType"
            minOccurs="0" maxOccurs="unbounded"/>
      </xs:sequence>
      <xs:attribute name="schema" type="xs:string" use="required"/>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

### 7.1.2 Efficiency Tests

The aim of the efficiency tests was to quantify the performance of the API methods of the CS service, more precisely of the functionality of the CS accessible via API.

The test environment was composed by Apache JMeter[8] 1.8.1 and Mozilla 1.3 [Mozilla/5.0 (Windows; U; Windows NT 5.0; en-US; rv:1.3) Gecko/20030312] on Windows 2000.

For each method of the CS API a test plan was created with the Apache JMeter test kit, simulating an XML-RPC request to the system. For these test suites it was determined how fast the system responded to each request. After that, for each API call, the system was subjected to a critical evaluation of its performance under high load, to determine its scalability. Therefore the test suits were started concurrently with an increasing number of simultaneous runs. Again the response times were determined. We report the min, the max and the average response time, and the error percentage of all tests. It was also checked how reliable the system was under the high load, by verifying the results and noting errors.

In the tables that follow are reported the results of all API tests, in terms of response time and error percentage.

Method: **addCollection()**

| conc. calls | loops | total | Avg ms | Min ms | Max ms | Error% | Rate |
|---|---|---|---|---|---|---|---|
| 1 | 200 | 200 | 166 | 130 | 1091 | 0,00% | 6,1/sec |
| 5 | 40 | 200 | 685 | 160 | 1442 | 0,00% | 6,6/sec |
| 10 | 20 | 200 | 1166 | 140 | 2774 | 0,00% | 6,1/sec |
| 15 | 15 | 225 | 9797 | 541 | 38085 | 0,00% | 1,4/sec |
| 20 | 10 | 200 | 2571 | 150 | 5779 | 0,00% | 5,5/sec |

[8] http://jakarta.apache.org/jmeter/

Method: **initializeCollections(Id,Name,Description,MembershipCond,UserId)**

| conc. calls | loops | total | Avg ms | Min ms | Max ms | Error% | Rate |
|---|---|---|---|---|---|---|---|
| 1 | 200 | 200 | 169 | 120 | 2444 | 0,00% | 6,3/sec |
| 5 | 40 | 200 | 727 | 130 | 5308 | 0,00% | 6,2/sec |
| 10 | 20 | 200 | 1700 | 140 | 12097 | 0,00% | 4,9/sec |
| 15 | 15 | 225 | 8884 | 30 | 24145 | 0,00% | 1,6/sec |
| 20 | 10 | 200 | 855 | 10 | 7841 | 0,00% | 5,8/sec |

Method: **listCollections()**

| conc. calls | loops | total | Avg ms | Min ms | Max ms | Error% | Rate |
|---|---|---|---|---|---|---|---|
| 1 | 200 | 200 | 111 | 80 | 441 | 0,00% | 9,0/sec |
| 5 | 40 | 200 | 395 | 80 | 1031 | 0,00% | 10,3/sec |
| 10 | 20 | 200 | 568 | 90 | 1822 | 0,00% | 9,9/sec |
| 15 | 15 | 225 | 4615 | 10 | 7651 | 0,00% | 2,9/sec |
| 20 | 10 | 200 | 230 | 90 | 1742 | 0,00% | 7,5/sec |

Method: **listCollections(userId)**

| conc. calls | loops | total | Avg ms | Min ms | Max ms | Error% | Rate |
|---|---|---|---|---|---|---|---|
| 1 | 200 | 200 | 154 | 120 | 531 | 0,00% | 6,5/sec |
| 5 | 40 | 200 | 621 | 130 | 1722 | 0,00% | 6,9/sec |
| 10 | 20 | 200 | 1017 | 140 | 2363 | 0,00% | 6,8/sec |
| 15 | 15 | 225 | 5698 | 30 | 10766 | 0,00% | 2,3/sec |
| 20 | 10 | 200 | 879 | 140 | 2914 | 0,00% | 6,1/sec |

Method: **getPersonalCollections(userId)**

| conc. calls | loops | total | Avg ms | Min ms | Max ms | Error% | Rate |
|---|---|---|---|---|---|---|---|
| 1 | 200 | 200 | 151 | 130 | 620 | 0,00% | 6,6/sec |
| 5 | 40 | 200 | 647 | 130 | 1262 | 0,00% | 6,6/sec |
| 10 | 20 | 200 | 1066 | 140 | 2664 | 0,00% | 6,7/sec |
| 15 | 15 | 225 | 1110 | 140 | 2772 | 0,00% | 6,1/sec |
| 20 | 10 | 200 | 1220 | 150 | 2824 | 0,00% | 5,8/sec |

Method: **editCollection(collectionMetadata,userId)**

| conc. calls | loops | total | Avg ms | Min ms | Max ms | Error% | Rate |
|---|---|---|---|---|---|---|---|
| 1 | 200 | 200 | 346 | 290 | 1092 | 0,00% | 2,9/sec |
| 5 | 40 | 200 | 1652 | 171 | 3144 | 0,00% | 2,8/sec |
| 10 | 20 | 200 | 3149 | 350 | 5729 | 0,00% | 2,8/sec |
| 15 | 15 | 225 | 5107 | 521 | 8022 | 0,00% | 2,6/sec |
| 20 | 10 | 200 | 5920 | 601 | 12338 | 0,00% | 2,4/sec |

Method: **getCollectionMetadata(collectionIds)**

| conc. calls | loops | total | Avg ms | Min ms | Max ms | Error% | Rate |
|---|---|---|---|---|---|---|---|
| 1 | 200 | 200 | 272 | 240 | 881 | 0,00% | 3,7/sec |
| 5 | 40 | 200 | 1353 | 260 | 2403 | 0,00% | 3,5/sec |
| 10 | 20 | 200 | 2653 | 260 | 4857 | 0,00% | 3,2/sec |
| 15 | 15 | 225 | 4010 | 391 | 8612 | 0,00% | 3,3/sec |
| 20 | 10 | 200 | 1659 | 270 | 5378 | 0,00% | 4,5/sec |

Method: **deleteCollection(collectionId)**

| conc. calls | loops | total | Avg ms | Min ms | Max ms | Error% | Rate |
|---|---|---|---|---|---|---|---|
| 1 | 200 | 200 | 91 | 60 | 1022 | 0,00% | 11,5/sec |
| 5 | 40 | 200 | 726 | 110 | 5888 | 0,00% | 5,6/sec |
| 10 | 20 | 200 | 608 | 70 | 4396 | 0,00% | 10,9/sec |
| 15 | 15 | 225 | 1835 | 130 | 8111 | 0,00% | 6,3/sec |
| 20 | 10 | 200 | 895 | 80 | 8802 | 0,00% | 6,5/sec |

Method: **deleteArchive(archiveId)**

| conc. calls | loops | total | Avg ms | Min ms | Max ms | Error% | Rate |
|---|---|---|---|---|---|---|---|
| 1 | 200 | 200 | 232 | 110 | 4647 | 0,00% | 4,8/sec |
| 5 | 40 | 200 | 1355 | 241 | 24756 | 0,00% | 4,8/sec |
| 10 | 20 | 200 | 3501 | 10 | 54057 | 0,00% | 4,6/sec |
| 15 | 15 | 225 | 6131 | 40 | 117780 | 0,00% | 1,7/sec |
| 20 | 10 | 200 | 851 | 120 | 3615 | 0,00% | 6,5/sec |

Method: **deleteUser(userId)**

| conc. calls | loops | total | Avg ms | Min ms | Max ms | Error% | Rate |
|---|---|---|---|---|---|---|---|
| 1 | 200 | 200 | 85 | 70 | 410 | 0,00% | 11,7/sec |
| 5 | 40 | 200 | 314 | 70 | 962 | 0,00% | 12,3/sec |
| 10 | 20 | 200 | 703 | 90 | 1853 | 0,00% | 9,1/sec |
| 15 | 15 | 225 | 838 | 70 | 2634 | 0,00% | 10,5/sec |
| 20 | 10 | 200 | 91 | 70 | 180 | 0,00% | 8,1/sec |

## 7.2 The Collection Service GUI

The graphical user interface of the CS is accessible via a web-browser. It has been designed keeping in mind the *easy-to-use* concept so it has been organized into two areas, the *menu area* and the *working area* as shown in figure 9. Menu area contains a *menu bar* (at the upper) and an *action menu*. Working area contains a *collection hierarchy area* and a *collection data area*.

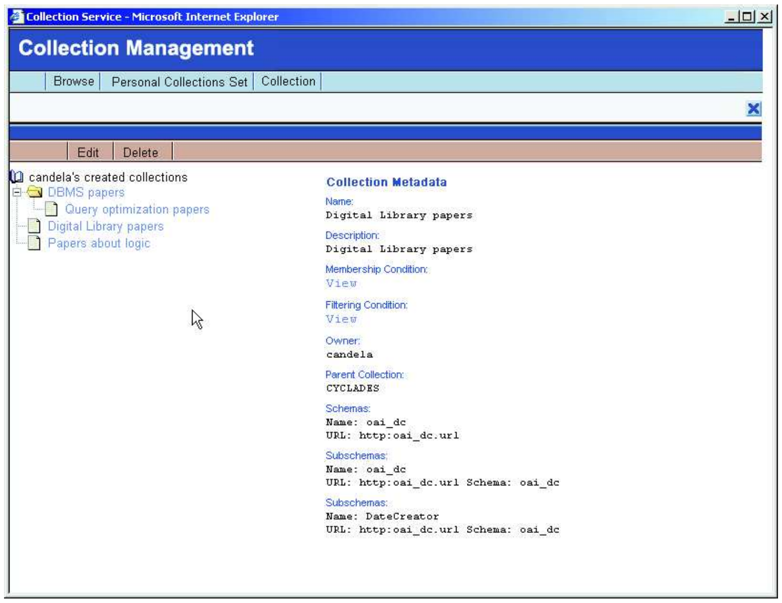
### The menu bar

At the upper of the interface (under the Collection Management title bar) there is a menu bar with three menus and/or action shortcut.

Via the *Browse* menu the user may choice the set of collections shown in the working area among own created collections and all CYCLADES collections.

Via the *Personal Collections Set* shortcut the user can browse/edit his "personal collection set". Figure 10 shows the GUI that allows user to manage his personal collection set. This GUI has a working area little bit different from the previous, there are two collections hierarchy areas, one (the left) for the "actual" personal collections set and the other (the right) for all collections. Clicking on a collection in the left area the user can remove this from the actual personal collections set, clicking on a collection in the right area the user can add this from the actual personal collections set. Collection data area in the middle shows collection data (e. g. name, description) for the selected collection.

Via *Collection→New* the user can create a new collection. The system will present a form (Fig. 11) to fill in, please enter here the name and the description of the new collection (useful to identify it later), the parent collection (collections may be organized hierarchically), and the membership condition (a CYCLADES query and/or one or more archives). The membership condition is the set of conditions that will characterize the set of documents belonging to the collection.



Collection Metadata

Name:
Digital Library papers

Description:
Digital Library papers

Membership Condition:
View

Filtering Condition:
View

Owner:
candela

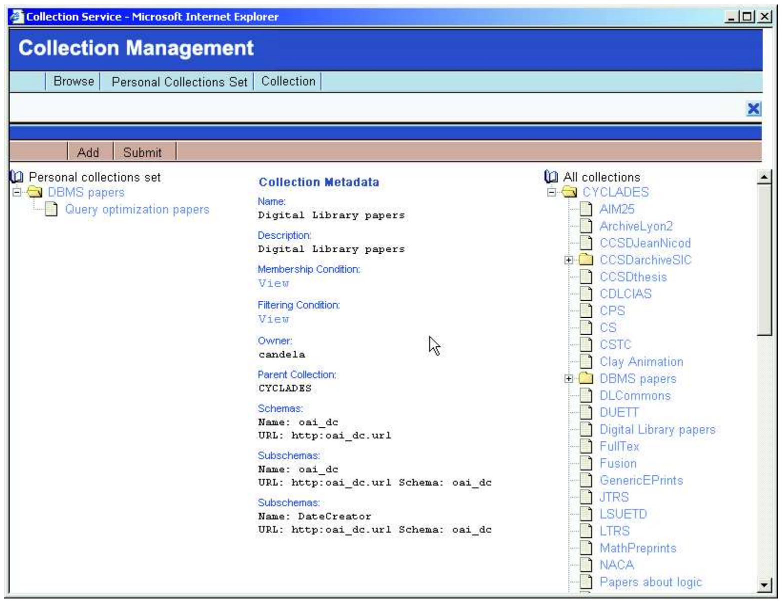
Parent Collection:
CYCLADES

Schemas:
Name: oai_dc
URL: http:oai_dc.url

Subschemas:
Name: oai_dc
URL: http:oai_dc.url Schema: oai_dc

Subschemas:
Name: DateCreator
URL: http:oai_dc.url Schema: oai_dc

Figure 9: CS Graphical User Interface.



Collection Metadata

Name:
Digital Library papers

Description:
Digital Library papers

Membership Condition:
View

Filtering Condition:
View

Owner:
candela

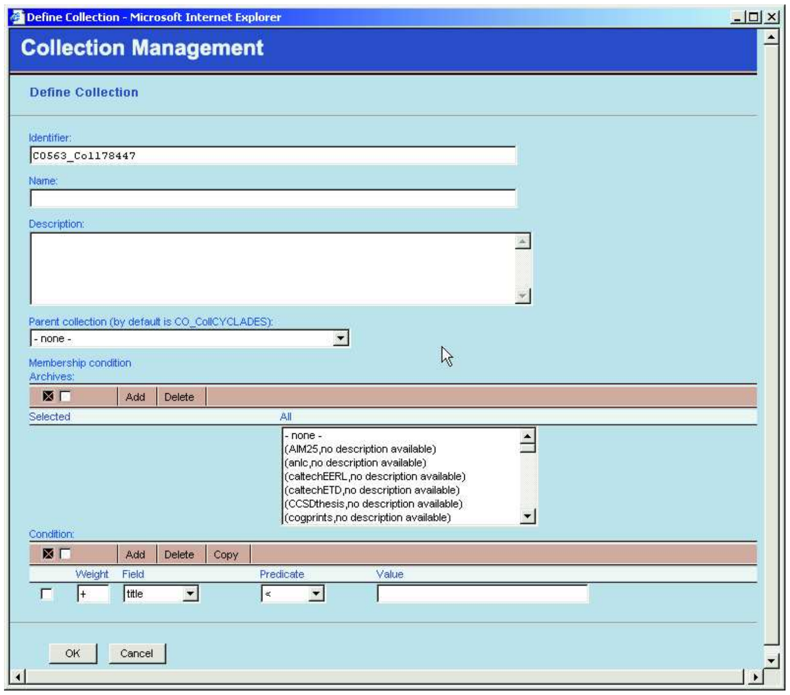
Parent Collection:
CYCLADES

Schemas:
Name: oai_dc
URL: http:oai_dc.url

Subschemas:
Name: oai_dc
URL: http:oai_dc.url Schema: oai_dc

Subschemas:
Name: DateCreator
URL: http:oai_dc.url Schema: oai_dc

Figure 10: CS Graphical User Interface for select the Personal Collections Set.



Identifier: C0563_Col178447

Name:

Description:

Parent collection (by default is CO_CollCYCLADES): - none -

Membership condition
Archives:

Selected

All
- none -
(AIM25,no description available)
(anlc,no description available)
(caltechEERL,no description available)
(caltechETD,no description available)
(CCSDthesis,no description available)
(cogprints,no description available)

Condition:

| Weight | Field | Predicate | Value |
|---|---|---|---|
| + | title | < | |

Figure 11: CS Graphical User Interface: Create Collection Form.

Interestingly, the system will automatically determine the best source from which to search for (this is the "Retrieval Condition").

### The action menus

Under the menu bar the CS interface provide an action menu. The items of this menu are related to the collection shown in the collection data area.

If the collection data area shows a collection created by the user than the action menu contains the item *Edit*, in order to edit this collection, and *Delete* in order to delete this collection.

### The collections hierarchy area

On the left of the working area there is the collections hierarchy area. In this area there is a navigable hierarchical view of the set of collections actually in use (own created collections or all collections).

Clicking on a collection allows a user to see collection data in the collection data area and, if the user has the rights, to manage them (via the action menu).

### The collection data area

On the right of the working area there is the collection data area. This area shows collection data (e.g. name, description) for the selected collection in the collections hierarchy area.

## 8 Related Work

In the DL field the concept of collection is broad, there is still confusion about what a collection is and what its characteristics are. In many papers, e. g. [9, 3, 15] the term collection is used as synonym of information source and the issue is how to automatically populate it. This paper has focused on collections as mechanisms for self-organizing the information space that a DL manages. However, we intend a collection as a virtual information source as it does not actually store any documents.

The concept of collection service proposed by Lagoze and Fielding in [12] shows many similarities with our CS mainly: (*a*) collection membership is defined through a set of criteria rather than containment and (*b*) CS must supply an independent mechanism for introducing meaningful and dynamic structure into a distributed information space. No implementation of this concept has ever be delivered.

Greenstone [15] propose an approach collection-centric where each collection has a user interface that allows users to search and browse over the collection. This kind of collection is similar to a IS, as the collection creator has to supply the documents belonging to it. This approach is quite static, the collection creator can add documents to a collection but has to do that manually.

In [8] the term "virtual collection" is introduced and a set of benefits for digital libraries that contains collections are outlined. That paper focuses on how to easily generate collection-level metadata without specifying how collection's documents have been collected and selected.

Many papers have been proposed about source selection in different fields. [16] proposes a database selection technique called TQRS for resolving the problem of query routing where the ISs are databases with multiple text attributes. That technique uses query sampling in order to acquire database's knowledge and then an extensions of the CVV ranking method [17] to rank each database. This is similar to the solution that we have proposed but we have used a revised version of CORI [5] instead of CVV because it is one of the most stable and effective [7] and compatible with resource descriptions acquired by query-sampling, while CVV is not [14].

## 9 Conclusion

This report has introduced the CYCLADES Collection Service, a service for supporting virtual collections. This service is exploited by other services to provide virtual views of the DL customized according to the needs of the different communities of users.

We consider the CS we described in this paper as a first prototype of a more general *mediator infrastructure service* that can be used by the other DL services to efficiently and effectively implement a dynamic set of virtual libraries that match the user expectations upon the concrete heterogeneous information sources. In particular, we are now working on the generalization of the Collection Service to include other aspects of the information space in the virtual view, like the structure of the documents, their descriptive metadata formats, and the used controlled vocabularies, i. e. terms used in describing a document.

**Acknowledgements.** This work was partially founded by the Cyclades project. Thanks are due to the DLib Group at CNR-ISTI for the considerable support received.

## References

[1] Dewey Decimal Classification. `http://www.oclc.org/dewey`.

[2] Z39.50 Maintenance Agency. `http://lcweb.loc.gov/z3950/agency/`.

[3] Donna Bergmark. Collection Synthesis. In *Proceeding of the second ACM/IEEE-CS Joint Conference on Digital Libraries*, pages 253–262. ACM Press, 2002.

[4] David C. Blair. The challenge of commercial document retrieval, Part II: a strategy for document searching based on identifiable document partitions. *Information Processing and Management*, 38:293–304, 2002.

[5] J. P. Callan, Z. Lu, and W. Bruce Croft. Searching Distributed Collections with Inference Networks . In E. A. Fox, P. Ingwersen, and R. Fidel, editors, *Proceedings of the 18th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 21–28, Seattle, Washington, 1995. ACM Press.

[6] Jamie Callan and Margaret Connell. Query-based sampling of text databases. *ACM Transactions on Information Systems (TOIS)*, 19(2):97–130, 2001.

[7] James C. French, Allison L. Powell, Jamie Callan, Charles L. Viles, Travis Emmitt, Kevin J. Prey, and Yun Mou. Comparing the performance of database selection algorithms. In *Proceedings of the 22nd annual international ACM SIGIR conference on Research and development in information retrieval*, pages 238–245. ACM Press, 1999.

[8] Gary Geisler, Sarah Giersch, David McArthur, and Marty McClelland. Creating Virtual Collections in Digital Libraries: Benefits and Implementation Issues. In *Proceedings of the second ACM/IEEE-CS Joint Conference on Digital Libraries*, pages 210–218. ACM Press, 2002.

[9] Greg Jane and James Frew. The ADEPT digital library architecture. In *Proceeding of the second ACM/IEEE-CS Joint Conference on Digital Libraries*, pages 342–350. ACM Press, 2002.

[10] Carl Lagoze, William Arms, Stoney Gan, Diane Hillmann, Christopher Ingram, Dean Krafft, Richard Marisa, Jon Phipps, John Saylor, Carol Terrizzi, Walter Hoehn, David Millman, James Allan, Sergio Guzman-Lara, and Tom Kalt. Core services in the architecture of the national science digital library (NSDL). In *Proceedings of the second ACM/IEEE-CS Joint Conference on Digital Libraries*, pages 201–209. ACM Press, 2002.

[11] Carl Lagoze and Herbert Van de Sompel. The Open Archives Initiative Protocol for Metadata Harvesting. `http://www.openarchives.org/...`

[12] Carl Lagoze and David Fielding. Defining Collections in Distributed Digital Libraries. *D-Lib Magazine*, November 1998. `http://www.dlib.org`.

[13] Carl Lagoze and Herbert Van de Sompel. The open archives initiative: building a low-barrier interoperability framework. In *Proceedings of the first ACM/IEEE-CS Joint Conference on Digital Libraries*, pages 54–62. ACM Press, 2001.

[14] Luo Si and Jamie Callan. Using Sampled Data and Regression to Merge Search Engine Results. In *Proceedings of the 25th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 19–26. ACM Press, 2002.

[15] Ian H. Witten, David Bainbridge, and Stefan J. Boddie. Power to the People: End-user Building of Digital Library Collections. In *Proceedings of the first ACM/IEEE-CS joint conference on Digital libraries*, pages 94–103. ACM Press, 2001.

[16] Jian Xu, Yinyan Cao, Ee-Peng Lim, and Wee-Keong Ng. Database selection techniques for routing bibliographic queries. In *Proceedings of the third ACM conference on Digital Libraries*, pages 264–274. ACM Press, 1998.

[17] Budi Yuwono and Dik Lun Lee. Server Ranking for Distributed Text Retrieval Systems on the Internet. In *Database Systems for Advanced Applications*, pages 41–50, 1997.