Providing Searchability Using Broker Architecture on an Evolving Infrastructure

Brouwer, P.M.; Brugman, H.; Kemps-Snijders, M.; Van der Peet, C.M.; Zeeman, R.H.M.; Kunst, J.P.

2014

Providing Searchability Using Broker Architecture on an Evolving Infrastructure

Matthijs Brouwer, Hennie Brugman, Marc Kemps-Snijders, Jan Pieter Kunst, Maarten van der Peet, Rob Zeeman, Junte Zhang

{matthijs.brouwer, hennie.brugman, marc.kemps.snijders, jan.pieter.kunst, maarten.van.der.peet, rob.zeeman, junte.zhang} @ meertens.knaw.nl

Meertens Institute, Joan Muyskenweg 25
1096 CJ Amsterdam, The Netherlands

October 24, 2013

Abstract

The Nederlab project aims to bring together all digitized texts relevant to Dutch history and language, both in terms of metadata and full-text content. Given that the data comes from a plethora of data providers, we present a technical solution to deal with the heterogeneity of datasets for access, which we call the *Broker*. It is an extra pivotal layer between the back-end and front-end of the data infrastructure to query and retrieve massive amounts of humanities data. Moreover, extra services can be embedded in the Broker, such as lexicon service for automated query expansion.

1 Introduction

The Nederlab project[1] aims to bring together all digitized texts relevant to the history of Dutch language and culture – consisting of terabytes of data – in one web interface for doing language and historical research, in particular for search and analysis. The initial dataset contains over 37 million records ranging from newspaper articles to complete books.

However, the complexity, heterogeneity in terms of valuable curated metadata and massive size of the different datasets consisting of terabytes is a major challenge. The number of datasets is set to increase. How to manage the querying and retrieval of the different datasets? Moreover, the evolving back-end of the infrastructure may change due to optimization and changing user requirements. Meanwhile, the whole infrastructure, including the front-end, should be allowed to progress and be kept stable in terms of code and functionalities. Therefore, we introduce an extra layer – the *Broker* service – in our infrastructure between the back-end and front-end, which acts as a stable mediator between both. From an information retrieval point of view, the *Broker* can increase coverage, improve the effectiveness of retrieval, and increase the ease of use.

In this paper, we give a technical description of this service within our humanities infrastructure. In Section 2, we give a concise description of the context of our work, in Section 3 the description of the *Broker*, and we conclude with our findings and discussion in Section 4.

2 Background

The Common Language Resources and Technology Infrastructure (CLARIN) initiative seeks to establish an integrated and interoperable research infrastructure of language resources and its technology.[2] The Nederlab project uses the Component MetaData Infrastructure (CMDI; [2]) as descriptive metadata or surrogates to store and retrieve relevant records as a pragmatic approach to achieve a certain degree of interoperability, which are textual resources from c. 800 to present. Given the heterogeneity of the metadata, we wrangle (munge) the metadata to ISOcat data categories (DC) stored in the Registry (DCR; [4]).

[1] See `http://www.nederlab.nl/`.

[2] See `http://www.clarin.eu/external/index.php?page=about-clarin`.

We can provide interactive and focused access to such diversity of records in a single search engine [7], however, scalability is an issue.
The idea of a *Broker* has been studied extensively within distributed information retrieval (DIR; [6, 3]), where a single *Broker* coordinates retrieval from many independent federated search services. For example, the effective use of Internet information has already been identified in 1995 as a major challenge given the rapid growth in data volume, user base, and data diversity and the use of brokers is presented as a solution [1]. While the role of a *Broker* is prominent in DIR, the results of [5] show that a *Broker* can also be a crucial component in aggregated search, where multiple nuggets of information are merged into one. This paper presents our solution to partly similar challenges for language resources and within the CLARIN and Nederlab contexts.

3 Broker architecture

Our *Broker* is a PHP based JSON webservice that can be accessed by the user interface to handle search requests. As illustrated in Figure 1, the webservice processes incoming requests as a subrequest to the relevant index and returns the response from this subrequest in a standardized format to its waiting client.

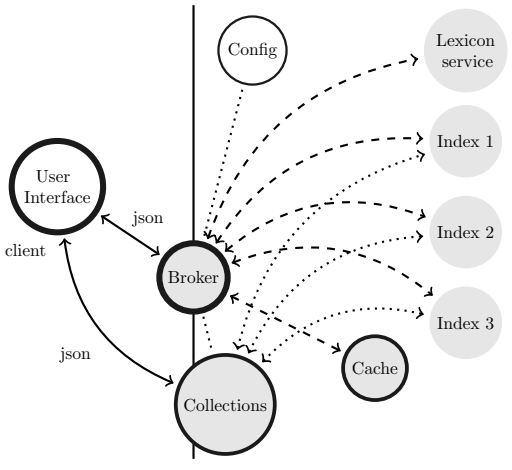

Figure 1: Broker Architecture providing searchability with multiple indexes, collections, caching and additional service in a cloud environment.

Information about the available Apache SOLR indexes and the functionality they provide has been made available to the *Broker* using configuration files. Essential part of this configuration is defining the set of field names to be used within the *Broker* and the mapping between these names and corresponding native names as used within each available index. Because the *Broker* uses these mappings to construct subrequests and translate subresponses when communicating with the indexes, a client only needs to know the set of field names to be used within the *Broker* when doing requests and parsing responses.

3.1 Processing requests

Derived directly from the syntax for some basic types of SOLR requests we minimally wanted the *Broker* to be able to handle, we introduced our own syntax for JSON requests. Using this syntax, a search request for first name *John* translates to a JSON request as in Figure 2 consisting of a *filter*, a *condition* and a *response*. As mentioned before, the field names used in such requests belong to the set of field names as used within the *Broker*.

```
{                                    "response": {
  "filter" : {                         "documents": {
    "condition": {                       "number": 10,
      "type": "equals",                  "start": 0,
      "field": "doctype",                "fields": [
      "value": "author"                    "id","firstname",
    },                                     "prefix","lastname"
  },                                     ]
  "condition": {                       }
    "type": "equals",                }
    "field": "firstname",
    "value": "John"
  },
```

Figure 2: Example JSON request searching for first name John *in* authors *and returning id, first name, prefix and last name for the first* 10 *results.*

On receiving a request and after some preliminary checks for e.g. valid JSON, the *Broker* starts by analyzing the request in more detail. First the index to which the actual request should be sent has to be determined. Therefore an inventory is made of all field names used in *filter*, *condition* and *response*. Based on this inventory and using the configured mappings,

the *Broker* computes whether or not a *single-index-solution* for this request is available, and which index should be used.
Having determined the index, the *Broker* continues by constructing a suitable query using the configured field name mappings for this index. This query is sent as a subrequest to the index, letting the *Broker* wait for a response. To increase performance, a caching mechanism for these queries is implemented.
The response on this subrequest is again analyzed by the *Broker*. Since for our Apache SOLR indexes the response is already JSON, basically only a translation of field names is performed using again the configured field name mappings for the index used for this subrequest. Finally, the translated response is sent back to the client.

3.2 Field name mapping

As mentioned in the introduction, for each index the *Broker* has a configurable mapping defined. These mappings allow each field name that can be used within the *Broker* to be mapped to one or more native field names within the index. The configured mapping is used in choosing an index, creating the request and translating the response on this request.
An illustrative example of a mapping can be found in Table 1. Here for each index, the *Broker* field name *id* is mapped to exactly one native field name. But as can be seen in the mapping of e.g. *sex*, a field name does not have to be mapped to every index. And the mapping of *name*, *firstname* and *lastname* shows that one-to-many and many-to-many relations are allowed.

Broker	Index 1	Index 2	Index 3
id	doc_id	id	author_id
sex	sex	female	—
firstname	DC-4194	—	author_firstname
lastname	DC-4195 last_name	lastname	author_lastname
name	DC-4194 DC-4195 last_name	—	author_fullname
...	...	...	...

Table 1: Example of mapping field names between Broker *and indexes.*

When applying the configured mappings in constructing the request, special attention from the *Broker* is needed for one-to-many mappings. In our syntax we require conditions to be constructed from elementary conditions bounded by possibly nested OR/AND conditions. If for some elementary condition a field name is involved that is mapped to several native equivalents, this condition has to be rewritten as to satisfy at least one of the new elementary conditions on these native equivalents, binding them with an OR condition. However, if the elementary condition contains a negation, binding of the new elementary conditions should be done using an AND condition instead.

In translating the response on a request, again special attention from the *Broker* is needed for one-to-many relations. For result lists we can have single native field names belonging to multiple *Broker* field names, but also multiple native field names belonging to the same *Broker* field names and combinations of both situations. The *Broker* handles these situations and tries to translate as much as possible to a result list containing only the required field names. Complex decisions have to be made when merging items from multiple native field names to one *Broker* field name, especially when facetting is involved. Therefore these one-to-many mappings should be avoided as much as possible and be made redundant by restructuring and rebuilding indexes once search requirements are well established.

As an additional property for a mapping between *Broker* field name and native field name, a *translator* may be defined. This allows us to deal with situations where compared to the *Broker* the native field name refers to items from another set. For example a *Broker* field name *sex* that may have values *male* or *female* mapping to the native field name *female* with values *true* or *false*. Since translation can be necessary both in constructing a request and rewriting a response, these translators should in general be applicable both ways. As with the complex mappings, the use of these translators should preferably be made redundant when restructuring indexes.

3.3 Collections

We extended the response object in our JSON request syntax with the option to store the result set as a *collection* and return only a collection identifying URL to the client. Typically such a stored set is nothing more than a list of identifiers for all items in the result. The returned URL, although only available temporary, can be used by a client in defining conditions for new requests. Alternatively, again by using the collection identifying URL, a client may download the *collection* and store it locally. Then whenever necessary a client can upload again this data to the broker, resulting in a new URL that also again temporarily can be used in requests later on.
Besides extending the *Broker* with the ability to temporary store and handle these collections, some plugins for Apache SOLR indexes had to be developed to facilitate these functionalities.

3.4 Joins

In our project the required search functionality did often appear to have a relational character, whereas our choice of indexes doesn't really provide this functionality. The introduction of *collections* however provides the client with an option to reuse results from previous requests as a condition to new requests, which effectively can be seen as a *join* operation.
We extended the JSON request syntax for the *Broker* by adding an optional *join* object to a *filter*. As illustrated in Figure 3, the existence of such a join lets the *Broker* split the requests in two separate subrequests, using the results from the first as a filter for the second.
Extending once more our request syntax, we also did allow filters with join to have a sub filter. This makes recursive appliance of joined filters in a single request to the *Broker* possible. Using these options a client can directly define even more complicated relational conditions without being bothered by technicalities like storing subresults or dividing into subrequests. However, to keep performance acceptable, usage of collections to implement the relational join has to be limited to situations where the size of such a collection is well below $1,000,000$ items.

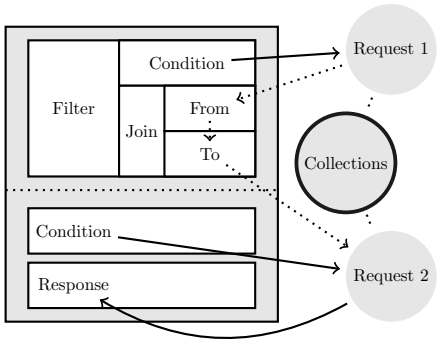

Figure 3: Broker splitting a request into two sequential subrequests, storing the first subresult as a temporary collection and using that as an additional filter condition for the second subrequest.

3.5 Lexicon Service

As can be seen from Figure 1, representing a scheme with the current *Broker Architecture*, we implemented the use of an external *Lexicon Service*. This provides the client with an option to automatically expand an elementary condition, i.e. a certain field having to equal a given string, to the new composite condition on this field to equal the given string or an variant as found with the *Lexicon Service*. Whenever the option is set, the *Broker* automatically handles calling the *Lexicon Service* and implements the result into an expanded request. Some special care has to be taken when negation is involved, but implementing this kind of functionality in the chain of parsing, creating subrequests and rewriting results is relatively easy.

4 Conclusion

The *Broker Architecture* makes it possible to keep terabytes of heterogeneous data manageable for search and retrieval. We use *sharding* to make the infrastructure scalable. It can improve coverage, effectiveness of retrieval, and increase the ease of use of a search engine for language researchers.
From a developer's point of view, it is possible to implement new infrastructural concepts with minimal disturbance for the user inter-

face. And as we explained with *collections*, *joins* and the *Lexicon Service*, our approach introducing a JSON request syntax that is parsed by the *Broker* makes it relatively easy to extend the service with additional functionality, such as further aggregation of currently separated records, and extra services helpful to language researchers such as an Ngram viewer. Using a directly configurable set of indexes and adaptive mapping of field names, the *Broker* offers further flexibility in constructing the infrastructure, and provides a relatively stable but easily extendable request syntax.
Although some limitations, particular in the use of collections and one-to-many mappings have to be considered, the *Broker* is a very workable solution for application developers for a continuous evolving research infrastructure.

Acknowledgments This work is supported by the Netherlands Organisation for Scientific Research (NWO), Royal Netherlands Academy of Arts and Sciences (KNAW), and Common Language Resources and Technology Infrastructure (CLARIN).

References

[1] C. Bowman, P. B. Danzig, D. R. Hardy, U. Manber, and M. F. Schwartz. The harvest information discovery and access system. *Computer Networks and ISDN Systems*, 28(1):119–125, 1995.

[2] D. Broeder, O. Schonefeld, T. Trippel, D. Van Uytvanck, and A. Witt. A pragmatic approach to XML interoperability the component metadata infrastructure (CMDI). In *Balisage: The Markup Conference 2011*, volume 7, 2011.

[3] F. Crestani and I. Markov. Distributed information retrieval and applications. In *Advances in Information Retrieval*, volume 7814 of *LNCS*, pages 865–868. Springer Berlin Heidelberg, 2013.

[4] M. Kemps-Snijders, M. Windhouwer, P. Wittenburg, and S. E. Wright. ISOcat: remodelling metadata for language resources. *IJMSO*, 4(4):261–276, 2009.

[5] S. Sushmita, H. Joho, and M. Lalmas. A task-based evaluation of an aggregated search interface. In *String Processing and Information Retrieval*, volume 5721 of *LNCS*, pages 322–333. Springer Berlin Heidelberg, 2009.

[6] P. Thomas. To what problem is distributed information retrieval the solution? *Journal of the American Society for Information Science and Technology*, 63(7):1471–1476, 2012.

[7] J. Zhang, M. Kemps-Snijders, and H. Bennis. The CMDI MI Search Engine: Access to Language Resources and Tools Using Heterogeneous Metadata Schemas. In *TPDL*, volume 7489 of *LNCS*, pages 492–495, 2012.