OD: Checkpoint 4

Franz Bermeo, Maryam Pashmi, Adrian Vogelsgesang

Our project is hosted under https://github.com/vogelsgesang/upc-od. We are using the wiki in order to organize our information. If one of the descriptions here is too superficious, more details with regards to the topics of this checkpoint can be found under https://github.com/vogelsgesang/upc-od/wiki/Checkpoint-2. **If necessary, the provenance of the documentation as well as the source coode can be tracked using the history function of this github respectively wiki. We are using http://trello.com as ticketing system.**

Changelog

- Meta data repository: added source::mapping and schema definitions
- GUI: possibility to edit the schema definitions
- Exposed direct access to the sources through our API
- GUI : Support for querying the sources individually
- Usability improvements for Marc21 adapter (better error checking)
- LibraryCloud API adapter implemented
- Implemented mapping of object types

Data sources

Name	Description	License	Format
Harvard Library Cloud	Bibliographic records of the Harvard university	CC0	JSON API
North Rhine-Westphalian Library Service Center's (hbz)	Linked data from Germany	CC0	JSON API
Cambridge University Library	Access to library catalogues	mixed	XML/JSON API
DBLP	On-line reference for bibliographic information on major computer science publications	ODC-BY 1.0 license	XML
CERN data set	Bibliographic Data from CERN	CC0	XML/Marc21
Open Library JSON API	spinoff of The Internet Archive	CC0 1.0	JSON
Cambridge Library	Bibliography Service	GPL	SPARQL
British Library	Bibliography Service	CC0 1.0	SPARQL

There are two main reasons why we chose our concrete datasources. On one hand side, there is the technology point of view. On the other side, we need a certain overlap, otherwise our deduplication algorithm would be pointless. We selected the CERN data set, because it provides data in Marc21/XML and there is a huge amount of legacy data out there which is coded in Marc21. By using Marc21 we make all this data usable. Harvard was selected because it provides a clean, well-documented and simple API. In addition, since both CERN

and Harvard are universities, we are expecting a overlap in the provided data. The British Library was selected due its vast amount of available data and because of its good documentation.

Cern Bibliography data (XML)

We decided to integrate the bibliography data of CERN available for bulk download (CC0 license). This data is coded as XML/Marc21, i.e. a recoding of the old Marc21 format to XML. The original Marc21 field names are still contained in the transcoded XML. Documentation for the meaning of these field names can be found under http://www.loc.gov/marc/bibliographic/ecbdhome.html. In our wiki we have a composition of all the relevant Marc21 field names.

In reference to data repository, we have chosen eXist-db as our native XML database. It provides a robust and efficient indexing to manage amounts of unstructured data, documents or collection. Thus we can take advantage of its support to XML queries and reduce the number of data transformations.

As technique to perform querying, we will use XQuery and its powerful features to manipulate XML-based object databases and in combination with XPath expressions to do much easier the surf through syntax. To achieve manage of the best way our data collections, we propose to use a well-known schema to represent the bibliography data and so to accelerate the evaluation of path expressions.

We are using eXistDB for accessing the XML sources. We can write XQuery-Documents and store them in the database. The results of evaluating these XQuery documents on the database are returned when sending a GET request to

```
<existDb-server>/exist/rest/<path of the document>?parameters
```

We can use placeholders in the XQuery document whose values can be specified as GET parameters. Hence, we can retrieve the data by querying the correct documents using the corresponding parameters. Alternatively, we can send a XQuery document to the relevant XML-document using a POST or GET request. eXistDb will return the results of evaluating this Xquery against the corresponding XML-document.

In both cases, the access to the database is achieved by sending HTTP requests to the eXistDB server. We are using the second possibility because the creation of the Xquery strings is more comfortable and more readable using an imperative language instead of Xquery. The resulting query is sent to eXist using a URL of the following form:

```
http://localhost:8080/exist/rest//db/od/books_export.xml?_query=/collection/...
```

The results are sent as XML by the eXistDb server. Hence, accessing the XML contents can be broken down to the task of accessing an XML API.

While trying to improve the performance by using indices, we found some bugs in eXistDb. We reported these bugs to the eXistDb team (Bug reports: 1, 2, 3). After circumventing these bugs, we were able to decrease the query time from 7 seconds to less than 1 second.

The Harvard LibaryCloud API (Json API)

We decided to use the Harvard LibaryCloud API which is offered under the Creative Commons Zero license (CC0) . This repository contains information from the Harvard Library Bibliographic Dataset, which is provided by the Harvard Library under its Bibliographic Dataset Use Terms and includes also data made available by, among others, OCLC Online Computer Library Center, Inc. OCLC Online Computer Library Center, Inc. and the Library of Congress.. This dataset contains over 12 million bibliographic records. The data is also available for bulk download. A more in depth documentation of the original data is available here.

Request to this API are sent as GET request offers a search functionality on top of these records and renames Marc21 fields into human readable names. In addition, the original Marc21 record is returned. We can execute 3 queries per second from a single IP address. Example query:

`http://librarycloud.harvard.edu/v1/api/item/?filter=keyword:internet`

The following excerpt shows some of the returned fields. Complete documentation is available on the web.

Field name	Field description
keyword	Keywords.
id	The identifier given to the item here in LibraryCloud. Exact matching.
title	The title and/or subtitle of the item. Exact matching.
title_keyword	The title and/or subtitle of the item. Keyword matching.

The API allows to filter on multiple criteria, and supports pagination and sorting the results.

Parameter name	Parameter description
filter	Filters. Syntax: fieldname:filter Multiple filter parameters can be provided.
limit	Number of records to return. Default is 25. Max is 250.
start	The starting point in the result set. Default is 0.
sort	Specifies the sort order. Default: “shelfrank desc”. *This parameter is undocumented and not officially supported!*

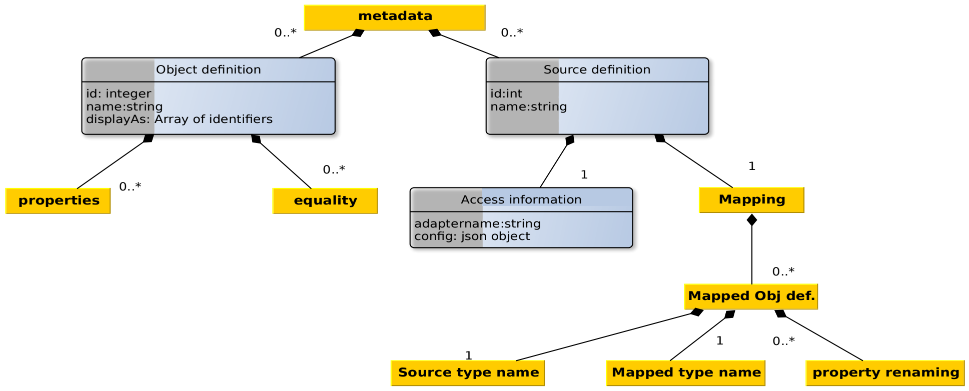
British library (RDF/SparQL)

We are integrating the data provided by the british library. This data is stored on servers of the British National Library and kept up-to-date by the library itself. It is made accessible through their SparQL endpoint http://bnb.data.bl.uk/sparql under the CC0 license.

We are going to write an adapter which is able to create the necessary Sparql query string and sends this query string to the SparQL endpoint http://bnb.data.bl.uk/sparql. The data returned by the British Library is formated as JSON and therefore easy to parse. We are trying to make this adapter general enough so that it can be reused for other SparQL services as well.

The documentation of the schema can be found under http://bnb.data.bl.uk/docs. A sketch of the schema is provided under http://www.bl.uk/bibliographic/pdfs/bldatamodelbook.pdf.

Meta data



Our internal schema is based on the **labeled** multi graph model: We have a set of entities (**books, persons, publisher, genre,...**) . Every object is an instance of a specific type and can have a set of properties (**title, publication_year, ...**). These entities can be linked by relations (**writtenBy, worksFor, isSequelOf, ...**) which are directed edges.

Properties on relations are not supported. Inheritance is not supported neither.

The specification of our schema is saved as a set of object type definitions. Next to a name and an enumeration of properties ~~and links~~, a definition of equality is stored which is used for the elimination of duplicates. **The differentiation between plain values and links is deferred until the actual data is retrieved. By postponing this distinction, we are eliminating the need to express the implicitly known type (link or plain value) explicitly and hence simplifying the configuration. In addition plain values might be converted to links by specifying a corresponding mapping for the respective field.**

In addition, we save information about our sources (how can the data be accessed? Which mapping must be done?) in machine readable form. All this information will be stored in a local MongoDB instance.

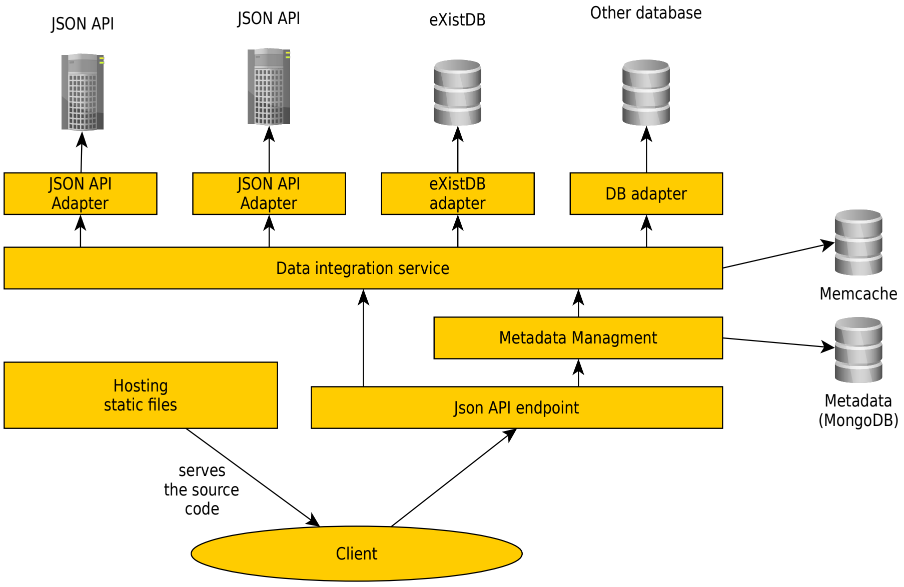
An example of a concrete configuration of a source can be found in the appendix.

Currently we are facing the following unsolved problems:

- Determining the equality of field values might be non-trivial. F.e the two author names "Rowling, J.K." and "Joanne K. Rowling" are referring to the same persons but a simple check for string equality will not unveil this equality. Even the usage of a Levenshtein distance does not solve this problem.
- Fields with different granularities. F.e.: some sources split the authors name in first name and surname, others only provide a field which contains both informations concatenated.
- Differences in the graph structure: Some sources store the author's name as a property of a book while others are adding a link to an author and store his name as a property of this linked node

Some thoughts about possible solutions for these problems and our mapping into a Mongo document can be found under https://github.com/vogelsgesang/upc-od/wiki/Meta-data

System architecture



Since most data will be retrieved from sources which are based on web technologies, our system will be based on web technologies, as well. It will be divided into a client and a server. The server is responsible for creating a consistent view of the data contained in the data sources. It provides access to this data by means of a JSON Api. Thanks to this, our software can be reused and integrated into other systems. The client makes calls to this Api in order to retrieve the data and display it to the end user in a more intuitive way. Additional Implementation details are provided in a separate wiki page.

The client

The client is a one-page-webapp which provides a GUI for the JSON Api of our server. It is served by our server statically, i.e. the server will not embed any type of information into the delivered Html/Js files. The JS code of the client is responsible for querying the relevant information using the server's Json API. It does not take care of any type of data integration.

The server

The server has four responsibilities:

- Deliver the clients source code
- Provide an API for modifying the meta data repository
- Build a consistent view of the data and deliver this view through the API
- Providing recommendations **(most likely we will use Random Walks; example: http://ids.csom.umn.edu/faculty/gedas/cars2010/bogers-cars-2010.pdf)**

The meta data is stored in a MongoDB instance. Access is provided using a REST Api. We are using mongoose (http://mongoosejs.com/) as a Object-Database-Mapping library and baucis (https://github.com/wprl/baucis) in order to expose the meta data as a REST endpoint. The data integration service is kept in sync by registering the corresponding mongoose callbacks on the models for the meta data. The corresponding REST Api is already implemented. During the implementation we came across a small bug in baucis (bug report, already fixed).

Consolidated data is served by constructing this data on the fly. For this purpose, all sources configured in the meta data repository are queried for relevant informations. This returned information is filtered and if its is actually fitting our initial query, we add it to the set of acknowledged information. Every time when new information is acknowledged, all other sources are checked again if additional informations can be found using the new set of available base information. This process is repeated until no source is able to contribute additional information.

All sources are connected with our data integration service through adapters (See Interface specification for source adapters). These modules are responsible for speaking to the relevant databases/web services. Every module delivers the data in a common format which still contains the field names of the source. The core of our data integration service takes care of mapping and restructuring these fields.

All sources are queried asynchronously, i.e. in parallel. This is done in order to avoid unnecessary idle times and for constructing the consolidated data faster. In order to further improve the response time, consolidated data is cached using MemCache. In addition, the unaggregated data returned by each source is cached in order to avoid the necessity of reloading the data from all sources if the configuration of one source is changed.

Functionality of the Data Integration Service

The data integration service must be able to serve two different types of queries:

First, there are queries by id: For these queries a set of ids of some of the relevant entries in a subset of our sources are already known. These Ids are always saved together with the id of the concrete source and are referring directly to the Ids of the bare original source. The integration service must retrieve the data for these ids and check if they are really referring to

representations of the same real world object. In addition it must search the sources for potentially existing additional representations of these objects. Finally, a combined view of the retrieved data is returned. This type of query needs to be answered for example when the end user/our recommendation algorithm follows a link within our consolidated view.

On the other side, there are also queries by attributes. In this case, the integration service is queried for example for all books with a specific title. Again, it has to combine multiple different representations of the same entity from different sources and provide a unified view of them. This type of queries will be used mostly directly by humans which are searching for a certain entity.

First, let us examine the algorithm for answering queries by ids. It is assumed, that the all records are already presented in the same schema. This common schema will be achieved by the field renaming. For the sake of understandability, this preceding schema mapping is ignored in the following sketch.

1. **Retrieve the records for the provided ids from the sources.**
2. **Combine retrieved data and initialize the set of known facts**
3. **Iterate until convergence:**
    1. **Ask all sources for additional relevant records based on the already known facts**
    2. **Combine retrieved data and update the set of known facts**

The first step is straightforward. The Ids are just forwarded to the relevant adapters which are responsible for retrieving the corresponding data records.
For the steps 2 and 3.2 we simply evaluate the equality rules defined in the meta data in order to check for identity between different records. As a performance optimization, the identity relations is assumed to be transitive and thereby it is avoided to check all pairs of source records for identity.
For step 3.1 the already known facts are used in order to build queries for the sources based on the identity defined in the meta data repository. These queries are given to the adapters in a disjunctive normal form. The DNF was chosen since it is a well-understood representation of logical equations and easy to manipulate. It is kept track of the queries which were sent to the sources in previous iterations of the loop in order to avoid querying the same source for the same conditions multiple times.
Convergence in step 3 is reached as soon as the queries to the source do not change anymore.

For the second type of queries (queries by facts) a similar algorithm is used which is sketched in the following:

1. **Query the data sources using the provided facts**
2. **Combine retrieved data and initialize the set of known facts**
3. **Iterate until convergence:**
    1. **Ask all sources for additional relevant records based on the already known facts**
    2. **Combine retrieved data and update the set of known facts**

Except for the initialization step, this algorithm is completely identical with the previous algorithm.

Appendix

Configuration of a source

Contents of file config-files/source-marc21.json

```
{"_id":"5358f92317bbf13036efbb71",
  "name":"Cern Bibliographic Data (Marc21)",
  "adapter":{
    "config":{
        "eXistEndpoint":"http://localhost:8080/exist/rest/",
       "xmlDocumentPath":"/db/od/books_export.xml",
       "limit":10
    },
    "name":"exist-marc21”
   },
  "mapping": [
    {
      "sourceType":"marcRecord",
      "mappedType":"book",
      "fieldMapping":{
        "language":["041","a"],
        "title":["245","a"],
        "author":["100","a"],
        "year":["260","a"],
        "publication_place":["260","a"],
        "publisher":["260","b"],
        "published_in_year":["260","c"],
        "isbn":["020","a"],
        "keyword":["653","a"],
        "topic":["650","a"]
      }
    }
  ]
}
```