OD: Checkpoint 5

Franz Bermeo, Maryam Pashmi, Adrian Vogelsgesang

During this project a web-based recommendation system is implemented. It is mostly focused on the recommendation of books, but it is one of the goals to keep the program general enough, so that it can be easily adapted to other scenarios and data sources.

Changelog

- Completely restructured this document
- Support for queries over multiple sources (no deduplication so far)
- GUI : Support for posing queries on the consolidated data view
- Usability improvements for LibraryCloud adapter (better error checking)
- SparQL adapter implemented
- Implemented mapping of field names

1. Introduction

As part of the course practice for OpenData, we are currently developing a recommendation system. Although we are investing most of our efforts in the recommendation of books and are covering in this document only book recommendations, we want to keep our system general enough in order to adapt it easily to other application domains such as movie or job recommendations.

Our main goals for this project are:

- A flexible, adaptable system architecture as stated above. In order to achieve this, we will rely heavily on our Meta Data repository.
- A realistic solution. This goal rules out especially the usage of artificial, non realistic data as for example contained in the training data set found under http://challenges.2014.eswc-conferences.org/index.php/RecSys#DATASET.
- A fast system. Recommendations should be provided nearly in real time. This implies that partial results instead of complete results are presented to the user.

We are using a RandomWalk algorithm for generating the recommendations. This algorithm is described in detail in section 2. Our internal schema as well as the schema of the meta data repository are described in section 3.

As data sources, we selected repositories which are available for free on the web. In concrete, we are relying on the bibliographies of CERN and Harvard as well as data provided by the British Library. The concrete criteria for the selection of these sources and their local data schemas are described in section 4.

After the descriptions of the involved concepts, algorithms and data sources, section 5 lays out the system architecture. Afterwards, section 6 will focus the data integration layer.

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. We are using http://trello.com as ticketing system.

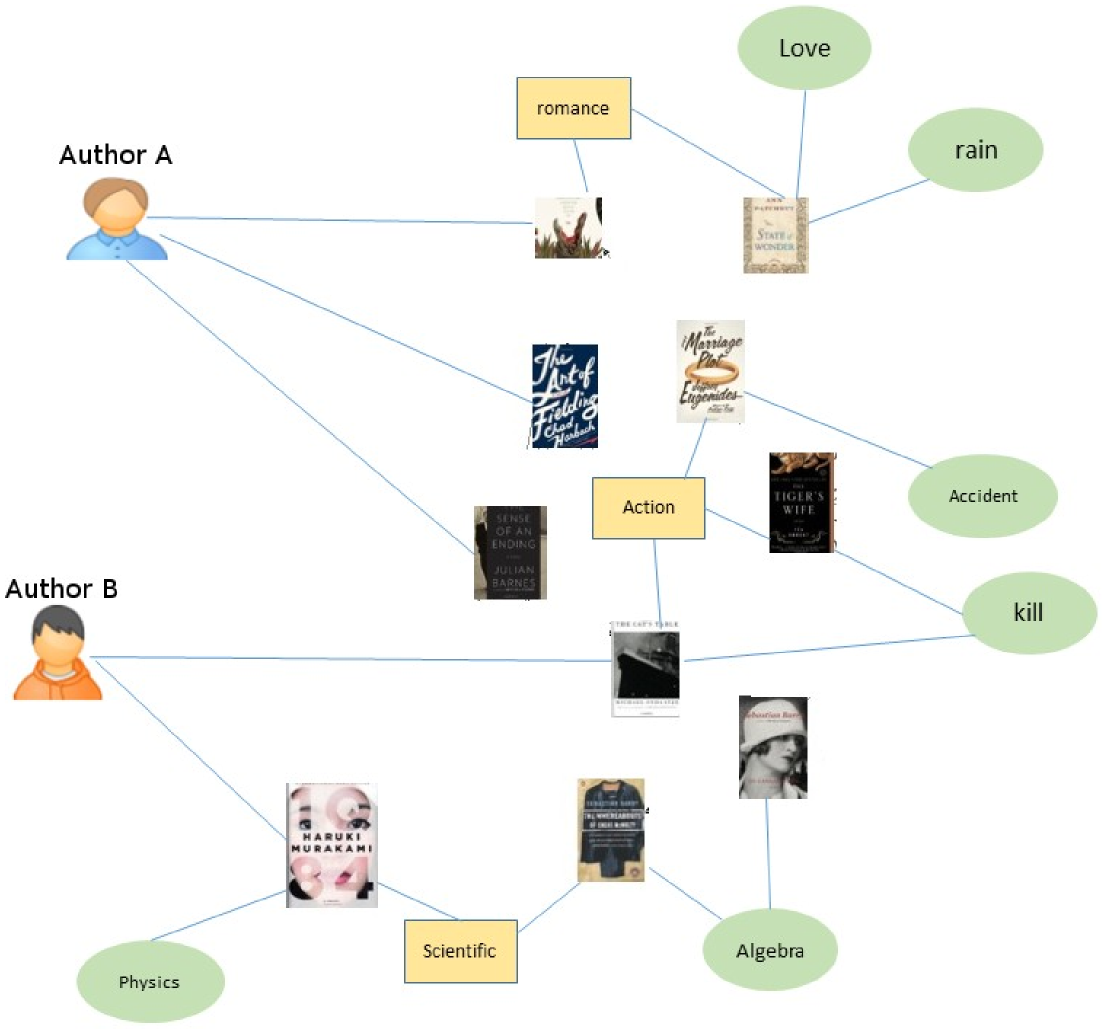
2. The Random Walk algorithm

For our recommendation system we will use the Random Walk algorithm which is based on Markov chains. Similar to the Google PageRank algorithm, recommendations are created using the topology of a graph. It is initialized based on the ratings of a user and returns other books which are closely linked to the already rated books.

This model estimate the transition probabilities from node to node using ratings and the links between different nodes.e.g.the authors associated with a book.

Our algorithm is based on the paper Movie Recommendation using Random Walksover the Contextual Graph which defines a Random Walk as:

“A random walk over G is a stochastic process in which the initial state is known and the next state S is governed by a probability distribution.We can represent this distribution for our graph G by constructing the transition probability matrix X, where the probability of going from node i to node j (at time t+1) is presentsd by Xi,j =P(St+1=j|St=i)”



In the other words, the algorithm starts at an randomly chosen initial node. Then, it jumps to one randomly selected neighbor of this current node. It stops as soon as it reaches a target node (in our scenario: a book). This random walk is repeated infinitely often and afterwards the books which were reached most often are recommended as highly relevant.

For our concrete recommendation algorithm we will focus on three types of objects: author, genre and publisher. A sketch of one of the used graphs can be found in the image above. The algorithm can be easily extended to additional features such as publication place or Amazon category without the need for retraining or changing the recommendation algorithm. The algorithm should be configurable using the MetaData repository.

While this paper suggests to calculate the resulting probability distribution exactly, this approach can be adapted easily in order to use sampled data by applying it to an incomplete instead of a complete graph of the domain. Hence, we can trade of recommendation quality against time. Furthermore, we have the opportunity to provide a first result quickly to the user and keep improving this recommendation as we are retrieving more and more parts of the underlying contextual graph.

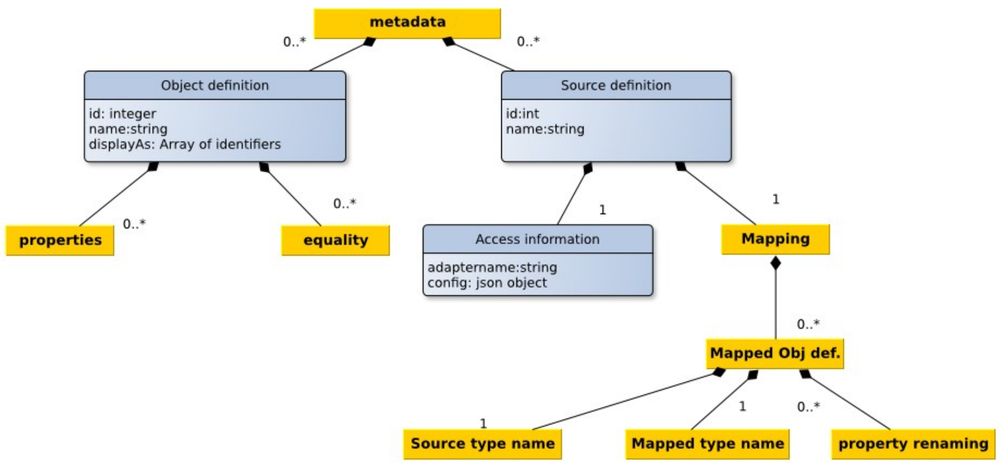
We limit our random walks to a finite depth and hence keep the user in the vicinity of his original book-related recommended need. By using ratings instead of binary usage patterns (as proposed in the original paper), we can bias the random walk to start from well rated book more probably than from lower rated books,instead of assigning each book the same starting point likelihood.

We are going to solve the cold start problem by using a standard set of ratings which provide a starting point for the Random Walk algorithm even if the taste of the user is not known so far. We will not replace this cold start recommendations immediately, as the user rates its first books. Instead, we mix this cold start recommendation with the user specific recommendations. The more books the user rated, the higher will be the weight of the user specific recommendations. As soon as the user provided enough input, we will replace the cold start recommendations completely by the user specific recommendations.

3. Schema of the Meta Data repository

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 attributes, 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.

After all, we are aware of 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

4. Data sources

We selected our data sources based on three main criteria: On one hand side, we took into account the provided data as well as the credibility of this data. On the other side, we based our decisions on the involved technologies. Additionally, we needed to assure a certain overlap of our data sources. Otherwise our deduplication algorithm would be pointless.

Cern Bibliography data (XML)

First, we decided to use the bibliography of the CERN research institute which is available for bulk download (CC0 license). The dataset is provided as a single XML file following the Marc21XML standard. This standard describes an encoding of the old Marc21 standard into XML. The original Marc21 field names are still contained in the transcoded XML. The documentation of these field names can be found in the original Marc21 specification.

We selected this source mostly in order to gain experience with the integration of a legacy data format into a new system. In addition, although Marc21 is not an up-to-date standard anymore, it is still a wide-spread standard. By supporting Marc21XML sources, our system can exploit the huge amount of data which is still coded in Marc21.

Marc21XML uses a quite simple schema: Basically, a Marc21 record is just an enumeration of fields. Each field might appear more than once as a part of a record. Fields are identified by numbers between 0 and 999. Each field contains subfields which are identified by lowercase letters. These subfields contain the actual values stored as strings. There is no metadata available.

The downloaded file is loaded into a local eXistDb server. This XML database server provides us indexing and facilitates the retrieval of the relevant parts of the CERN's bibliography. The concrete queries are posed as Xqueries. These queries are built automatically (mostly using string concatenations) and are hidden from the end user. The Xquery is dispatched by sending it to as part of an HTTP request to the REST interface provided by eXistDb. For example, the URL

http://localhost:8080/exist/rest//db/od/books_export.xml?_query=<query>

returns the results of evaluating the provided Xquery against /db/od/books_export.xml.

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 0.01 second.

Harvard Library Cloud API (Json API)

As second data source we are using the Harvard LibaryCloud API. This API provides access to the bibliographic data from the Harvard Library Bibliographic Dataset, which is provided by the

Harvard Library 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. The data is made available under the CC0 license.

This API provides a wrapper over the underlying Marc21 records of the Harvard university. These Marc21 records is also available for bulk download. A more in depth documentation of the original data is available here.

Furthermore, this source provides rankings of the contained book resources based on the circulation data collected by Harvard. This additional circulation data is the actual reason why we are using this API instead of the bulk download. We are planning to incorporate this circulation data into our recommendation system. In addition, the Harvard LibraryCloud API provides a clean, well-documented and simple API. In addition,since both CERN and Harvard are research facilities, we are expecting an overlap in the provided data.

From a technical point of view, the Harvard API is an usual Json API hosted under http://librarycloud.harvard.edu/v1/api/item/. It allows filtering, pagination and sorting using GET parameters. For example, a query could have the following URL:

http://librarycloud.harvard.edu/v1/api/item/?filter=keyword:internet&start=20&limit=10&sort=score_downloads%20desc

The schema is again pretty simple: one JSON document per book. Again, no machine-readable metadata is available.

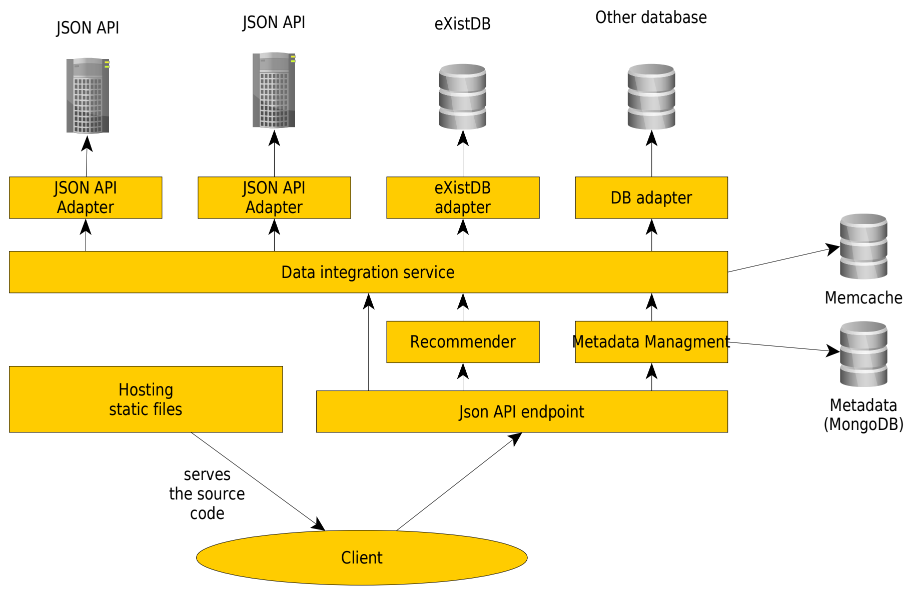
British Library (RDF/SparQL)

As third data source, we are incorporating 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 a 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 striving to make this adapter general enough so that it can be reused for other SparQL based services, as well. By implementing a generalized SparQL adapter we make sure that our software will be able to use modern data sources 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.

5. 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

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. The concrete algorithm will be described in more details in section 6.

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.

Aside from exposing this consolidated data directly through the Json API, our recommendation algorithm also uses this data in order to create the recommendations. Therefore, the consolidated graph is constructed in memory and the Random Walk algorithm is applied based on this graph

6. Data integration layer

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 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 all records are already available in the global schema. For the sake of understandability, the 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 conditions defined in the meta data repository. 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 which basically only differs in the initialization step and is otherwise completely identical to the previous algorithm.

Instead of the above algorithm, our actual implementation exploits even more parallelism by not only asking all sources in parallel (as sketched above) but by continuing with the next iteration of the loop as soon as one of the sources answered (instead of waiting for all answers as stated above). Furthermore, multiple different objects can be requested in parallel. This ability is used for example in order to speed up the construction of the graph for the recommendation 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"]
      }
    }
  ]
}
```