DOI: 10.5281/zenodo.5702427

Transforming Research through Innovative Practices for Linked Interdisciplinary Exploration

[September 2021]

Advancing Open Scholarship

D5.2 – Report on the Recommender System

Version 1.0 – Final

PUBLIC

H2020-INFRAEOSC-2019

Grant Agreement 863420

The project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 863420

D 5.2 Recommender System

Project Acronym:	**TRIPLE**
Project Name:	**Transforming Research through Innovative Practices for Linked Interdisciplinary Exploration**
Grant Agreement No:	**863420**
Start Date:	**1/10/2019**
End Date:	**31/03/2023**
Contributing WP	WP5, WP4, WP3
WP Leader:	Luca de Santis
Deliverable identifier	D 5.2
Contractual Delivery Date: 09/2021	**Actual Delivery Date:** 09/2021
Nature: Report	**Version: 1.0 Final**
Dissemination level	**PU**

Revision History

Version	Created/Modifier	Comments
0.0	Luca de Santis	Template
0.1	Simone Kopeinik	Deliverable Structure and Content Overview
0.2	Robert Thomann	Technical Description
0.3	Dieter Theiler, Robert Thomann	Technical Details
0.4	Dominik Kowald, Emanuel Lacic	Details on Recommender Integration

0.5	Simone Kopeinik	Revise Technical Description
0.6	Simone Kopeinik	Overview, Sections 1 and 2
0.7	Simone Kopeinik	Sections 5 and 6
0.8	Luca de Santis, Dieter Theiler	Review of the Document
1	Simone Kopeinik	Feedback incorporated

Table of Contents

Table of Figures

Acronyms

EOSC	European Open Science Cloud
RI	Research Infrastructure
RS	Recommender System
ScaR	Scalable Recommendation-as-a-service
MP	Most Popular
CF	Collaborative Filtering
CBF	Content-based Filtering
CCF	Hybrid Recommendations
REST	Representational state transfer
GUI	Graphical User Interface

Publishable Summary

This deliverable describes the Recommender System in GoTriple. It gives an overview of the background and of the work done so far in Task 5.2.

The document is structured as follows: after a brief introduction into the topic that is given in Section 1, a summary of how the Recommender System service is applied in GoTriple and what benefits it provides to the user can be found in Section 2. Section 3 gives an overview of ScaR – Scalable Recommendation-as-a-service and its software architecture. ScaR is a novel recommender framework following the *Software-as-a-Service* paradigm. It forms the foundation of the GoTriple recommendation system. How ScaR is used and integrated in GoTriple is presented in Section 4. This includes a fairly detailed description of REST-API services as well as recommendation algorithms and configurations (i.e., profiles) that have been implemented so far.

Section 5 outlines the current state of the work done in TRIPLE. As this is only the first stage of the Recommender System development in GoTriple, we give an overview of planned future work and focus directions in Section 6.

1 DESCRIPTION OF THE SERVICE

With GoTriple we build a platform that harvests different sources of scientific data collections and provides researchers with a unique access point for this information. Task 5.2 is adding to the platform's user friendliness, focusing on the exploration and integration of tailored information through ranking algorithms implemented in form of a Recommender System.

Recommender Systems (RSs) are software components that predict and suggest items of interest to users or user groups. Dedicated software services, which are typically embedded in websites, exploit user data to ease the information overload or act as sales assistants, supporting decision-making processes and gain customer loyalty [1]. RS have become quite popular being commonly integrated in all kind of information providing websites. Most prominent examples of Recommender System applications include companies, such as Amazon [2] or Netflix [3].

While initial recommendation approaches mainly exploited user preferences based on explicit ratings, later approaches explored a greater variation of user interactions to train recommendation models within observed environments [4]. To date, recommendation algorithms mostly use data about users, items and user-item interactions, where users' historical data traces form the main information source of the recommendation logic [5]. This means that algorithms continuously predict users' preferences and needs based on choices they made in the past. To deal with large data volumes and the constant embedding of user interactions, sophisticated models, methods and technical frameworks are required to cope with respective data while collecting, analysing, providing and predicting information in real-time [6].

Based on the ScaR – Scalable Recommendation-as-a-service framework our aim is to offer an intelligent and GoTriple tailored bundle of hybrid recommendation services, extending on collaborative and content-based recommendation approaches. To this end, we collect, model and exploit a rich amount of user and content data to suggest valuable diverse and interesting research documents, research projects and research peers.

While the full integration with GoTriple is still work in progress, we follow a continuous implementation approach updating the interfaces, algorithms and database structures on availability of new data sources and user study insights.

To incorporate the user needs, the task will benefit from the results of T3.3, where workshops are conducted in order to understand what GoTriple users consider most beneficial in recommendation services.

In accordance with the GoTriple development, we will implement an A/B testing approach. With this, the recommendation services can be evaluated and tailored to the specific needs of researchers who are visiting the GoTriple platform. This will form the final adaptation and selection of recommendation algorithms.

2 THE SERVICE IN TRIPLE

While GoTriple provides researcher with an amazing opportunity to explore and connect within different research fields, the overload of information may lead to the need of additional guidance and personalization. GoTriple's Recommender System (RS) is a service that supports researchers in finding their items of interest and with this enhances their discovery experience.

The final version of the GoTriple Recommender System will offer recommendation strategies that suggest research documents, research peers, and research projects to the user. A simplified depiction of the platform integration is illustrated in Figure 1.

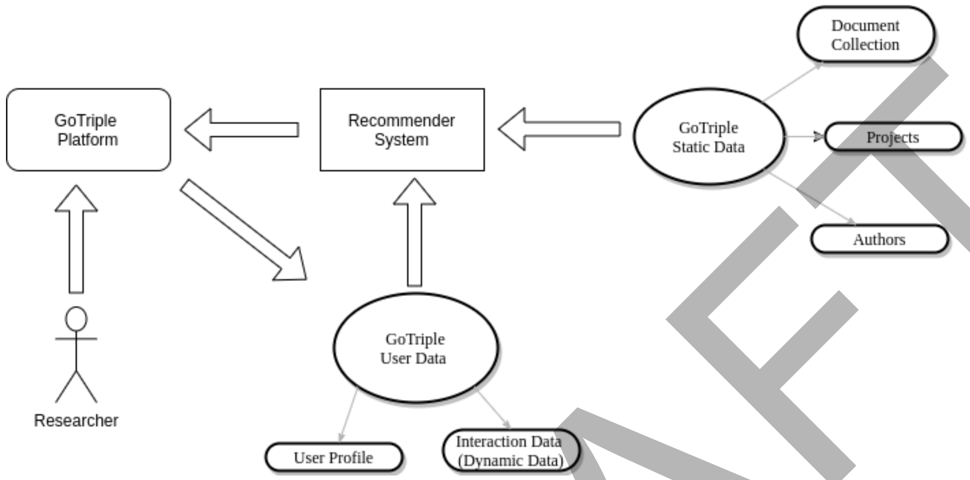
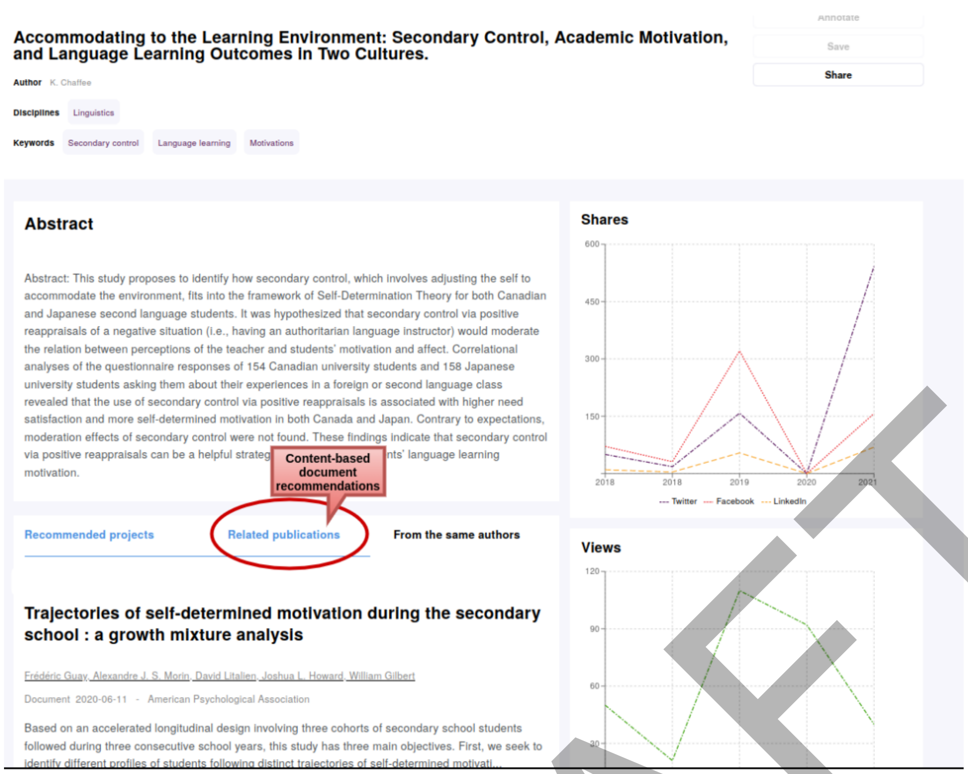


FIGURE 1: INTEGRATION OF RECOMMENDER SYSTEM IN GOTRIPLE

The RS is based on static data available in the system (i.e., meta-data of documents, research projects and research peers) and in addition, is continuously informed with emerging user profile and user interaction data. GoTriple user data is collected in the platform and sent to the RS on occurrence, in an event-based manner. This data is integrated and taken into account by the recommendation services in near-real time. Triggered by the navigation of the user in the platform, GoTriple calls respective recommendation service endpoints to retrieve a list of suggested items that are then displayed in the GoTriple GUI.

Figure 2 presents a screenshot of document recommendations in GoTriple. In line with the GoTriple GUI design, the RS provides two options of research document recommendations: (i) Content-based document recommendations based on document properties and (ii) personalized document recommendations based on the user's past interactions with GoTriple. The illustration shows option (i) which is embedded in the main document view. The recommendations are requested and presented to users when the "Related Publication" tab is selected. Option (ii), recommendations tailored to the user preferences, can be found in the personal page of the user but in later stages might also be available in a sidebar.



Accommodating to the Learning Environment: Secondary Control, Academic Motivation, and Language Learning Outcomes In Two Cultures.

Author K. Chaffee

Disciplines Linguistics

Keywords Secondary control Language learning Motivations

Abstract

Abstract: This study proposes to identify how secondary control, which involves adjusting the self to accommodate the environment, fits into the framework of Self-Determination Theory for both Canadian and Japanese second language students. It was hypothesized that secondary control via positive reappraisals of a negative situation (i.e., having an authoritarian language instructor) would moderate the relation between perceptions of the teacher and students' motivation and affect. Correlational analyses of the questionnaire responses of 154 Canadian university students and 158 Japanese university students asking them about their experiences in a foreign or second language class revealed that the use of secondary control via positive reappraisals is associated with higher need satisfaction and more self-determined motivation in both Canada and Japan. Contrary to expectations, moderation effects of secondary control were not found. These findings indicate that secondary control via positive reappraisals can be a helpful strateg nts' language learning motivation.

Trajectories of self-determined motivation during the secondary school : a growth mixture analysis

Frédéric Guay, Alexandre J. S. Morin, David Litalien, Joshua L. Howard, William Gilbert

Document 2020-06-11 - American Psychological Association

Based on an accelerated longitudinal design involving three cohorts of secondary school students followed during three consecutive school years, this study has three main objectives. First, we seek to identify different profiles of students following distinct trajectories of self-determined motivati...

FIGURE 2: SCREENSHOT OF GOTRIPLE SHOWING CONTENT-BASED DOCUMENT RECOMMENDATIONS. THE TAB IS MARKED IN RED

In line with the progress in platform development (e.g., availability of meta-data) at the time only document recommendations are fully implemented and available within the GoTriple platform.

However, the software infrastructure for the recommendation of research project and research peers is already in place and ready to be adapted.

The next two Sections provide a detailed technical description of the Recommender System, its underlying software framework ScaR (Section 3) and its application in Triple (Section 4).

3 SOFTWARE ARCHITECTURE

The system architecture used in the GoTriple Recommender System (RS) is based on the scalable recommendation framework ScaR. This section gives a technical overview of ScaR's software design, its deployment and configuration options.

3.1 Recommender System

In a nutshell, ScaR is composed of six modules. It implements the Microservice[1] architectural design pattern that structures the application as a set of loosely connected but collaborating modules. Communication is implemented with REST-based webservices. This supports a high level of maintenance and testability [7]. ScaR's general composition is depicted in Figure 3.

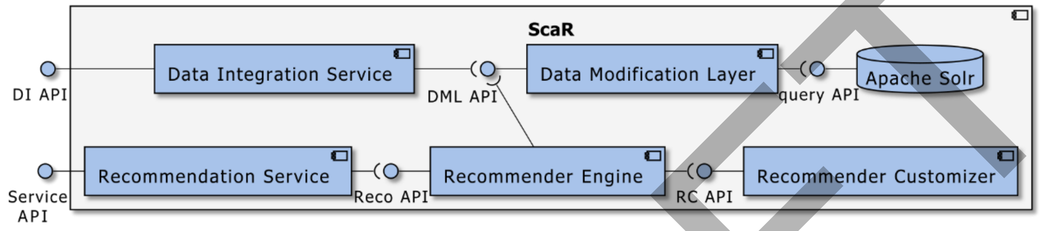


FIGURE 3: SCAR COMPONENTS

The following briefly elaborates on the function of the modules and their interconnections to each other: the Recommendation Service and the Data Integration Service provide interfaces to external systems. The Recommendation Service implements an interface for requesting recommendations. These recommendations are computed in the Recommender Engine which forms the central part of the framework. Using the Recommender Customizer module, it is possible to adapt the recommender to the domain data through the use of customization profiles.

The Data Integration service provides the API for pushing item meta-data and user interaction data to the RS. The Data Modification Layer handles all communication (i.e., storing and querying data) with the underlying data backend Apache Solr 8 that serves as the main backend data storage.

The services are implemented in the Java programming language and use either the open-source framework Spring[2] or Dropwizard[3]. For deployment they are packaged as Docker images, which enables operating system level virtualization.

The following Subsections describe the modules in greater detail.

[1] https://en.wikipedia.org/wiki/Microservices
[2] https://spring.io/
[3] https://www.dropwizard.io/en/latest/

3.1.1 Recommendation Service

The Recommendation Service provides RESTful services for requesting recommendations. Together with the Data Integration Service, it functions as the main entry- and communication -point between the platform and the RS. When the Recommendation Service is queried for recommendations, the query is parsed and then sent to the Recommender Engine. Finally, the Recommender Engine's response is delivered to the caller. Figure 4 shows this process of generating a recommendation.

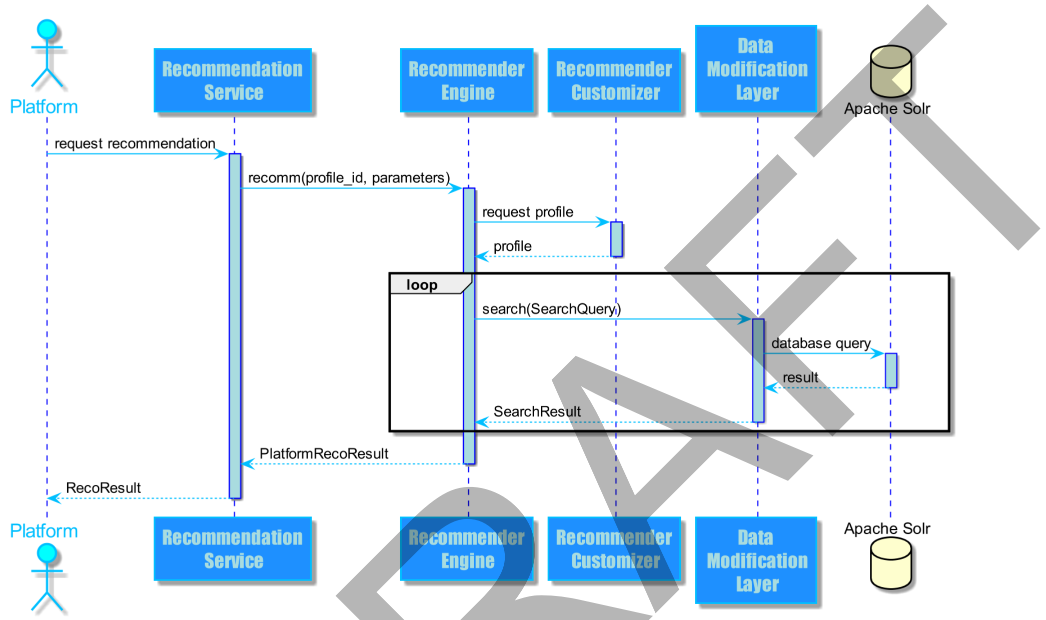


FIGURE 4: SEQUENCE DIAGRAM FOR RECOMMENDATION.

3.1.2 Recommender Engine

The Recommender Engine is the centrepiece module of ScaR since its purpose is to calculate recommendations. Herein Apache Solr's built-in data structures are being leveraged to enable efficient similarity calculation. The Recommender Engine supports standard approaches like collaborative and content-based filtering as well as hybrids between them. In addition, other algorithms can be added as needed, depending on the particular use case. The Recommender Engine generates recommendations requested by the Recommendation Service as shown in Figure 4. For this, it retrieves the recommendation profile from the Recommender Customizer, generates the required search queries, queries the Data Modification Layer, and finally composes the recommendation result.

IMPLEMENTED RECOMMENDER ALGORITHMS

The RS supports the following recommendation algorithms:

- **Most Popular (MP):** This is an un-personalized algorithm. MP recommends to each user the same set of items, which are weighted and ranked by their popularity (e.g., the item which was accessed the most).
- **Collaborative Filtering (CF):** This is a personalized algorithm that analyses the interaction data on items to find similar users, and then recommends items of these similar users [9]. In particular, CF finds the nearest neighbours based on some similarity metric, and then recommends items that the neighbours have liked but the target user still has not consumed.
- **Content-based Filtering (CBF):** This also is a personalized algorithm that calculates item-similarities based on content features (e.g., title, full text, keywords, ...) and then recommends these similar items [10]. Content-based recommendation systems analyses item meta-data to identify other items that could be of interest for a specific user. This can be done based on user profile data or on the meta-data of the items that the user has liked or purchased in the past. Our implementation of the content-based recommender utilizes similarity queries directly supported by the database backend and allows to tune parameters for respective built-in similarity measures, if needed.
- **Hybrid Recommendations (CCF):** All three mentioned recommender algorithms have unique strengths and weaknesses, e.g., CF suffers from sparse data and cold start problems, while content-based approaches strongly depend on the quality of meta-data to be utilized. Hybrid recommenders combine different algorithms to tackle this issue in order to produce more robust recommendations. Considering that we want to favour items recommended by more than one method, we chose to implement a hybrid approach called Cross-Source Hybrid defined in [11], where the combined weighting of the recommended *item i*, is given by the sum of all single weightings from within each recommender method, multiplied by the weights of the recommender methods themselves. This combination can be further extended by theory-informed models of the problem space.

3.1.3 Recommender Customizer

The Recommender Customizer module holds customization profiles for each recommendation algorithm, which allow to adapt the recommender to the domain data. Here, it is specified what is recommended, which features are considered, the adjustment of individual input parameters (e.g., the number of recommended items) and how recommendation algorithms shall be applied in principle. For example, we could define the similarity measure (e.g., Jaccard or Cosine) in case of CF or the weightings of used recommender methods in case of hybrid approaches. The Recommender Engine automatically takes into account respective recommender profiles to directly affect the calculation of recommendations. Figure 5 shows a typical recommendation profile and depicts the common hierarchical structure defined with the help of the YAML[4] data serialization language [13]. Currently, ScaR supports 14 different types of recommendation profiles.

[4] https://en.wikipedia.org/wiki/YAML

```
!!at.knowcenter.sc.recomm.common.profiles.InteractionCfProfile
recommType: itemscoreNames:
  item: items
recommProps: true
fetchUserHistoryProperties:
  fetch: true
  itemCount: 10
  interactionCoreProperties:
    interactionCount: 40
    timeFieldName: timestamp
    userFieldNames:
      - user_id
      - session_id
    itemFieldName: document_id
    interactionTypes:
      - TAG_DOCUMENT
      - OPEN_DESCRIPTION
      - ACCESS_DOCUMENT
      - EXPORT_INFORMATION
similarUsersProperties:
  similarUserCount: 40
  topKSimilarUserCount: 40
  minItemOverlap: 1
similarItemsProperties:
  filterUserItems: true
  flexFilters:
    fieldFilters:
    - fieldName: type
      fieldValue: DOCUMENT
  interactionFields:
    fields:
    - name: users_tag_document
    - name: users_access_document
    - name: users_export_information
    - name: users_open_description
  usersToIgnore: []
```

FIGURE 5: A TYPICAL RECOMMENDER PROFILE.

3.1.4 Data Integration Service

The Data Integration Service provides RESTful APIs for uploading data and user interactions to the backend database. Together with the Recommendation Service, it functions as the main entry- and communication-point between the GoTriple platform and the RS.

Two types of data objects are ingested by the Data Integration Service: items, and user interactions.

Items are ~~all~~ entities who are recommended to users. Typically, items are posted to the Data Ingestion's REST interface, processed, and sent to the Data Modification Layer for storage in the database as depicted in Figure 6.

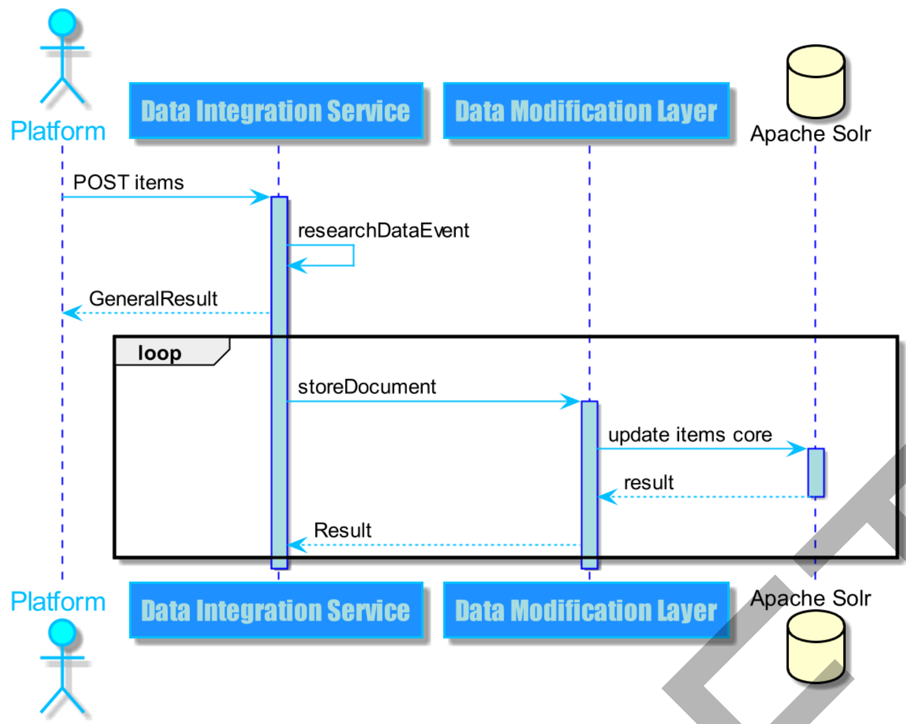


FIGURE 6: INGESTION OF ITEMS.

Figure 7 shows the ingestion process for user interactions. User interactions are posted to the Data Integration Service's REST interface. When interaction data is pushed to the Data Integration Service, then the service validates the interaction data. If the interaction is valid the service executes the required database updates: this includes at least the storing of the interaction itself but, depending on the type of interaction, other database entries (e.g., items) must be updated as well. The resulting database update requests are then sent to the Data Modification Layer and stored in the database.

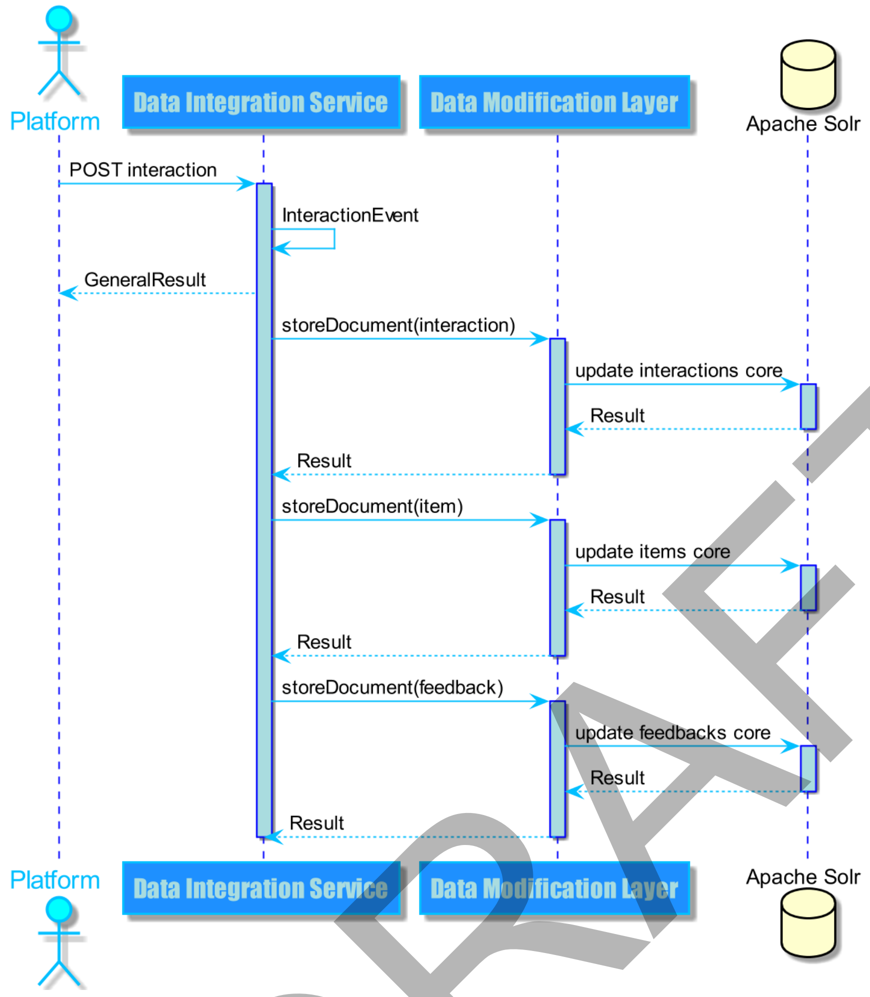


FIGURE 7: INGESTION OF A USER INTERACTION.

3.1.5 Data Modification Layer

The Data Modification Layer (DML) serves as data transfer intermediary between the individual ScaR modules and performs CRUD (create, retrieve, update and delete) operations in interaction with the Apache Solr database. It is responsible for encapsulating the underlying data storage and can be adapted accordingly. This data backend solution not only guarantees scalability and near real-time recommendations but also the support of multiple data sources. The DML makes extensive use of these search capabilities to increase the system performance.

3.1.6 Apache Solr

ScaR uses Apache Solr [5] as an underlying database. In principle, ScaR could be extended to work with different document-based database engines such as Elasticsearch[6]. Currently, Solr has been selected as the database because it offers two major advantages for recommendation use cases. The first is its query speed [11], which is most important for near real-time applications such as RS as they demand a fast execution of the search operations. The second is its ability to work seamlessly with different types of entities. Solr schemas, which define how a core/collection (i.e., document index) stores data, are very flexible, and easy to extend via configuration files. Besides, Solr can have multiple cores, which allows for storing data with different structures and provides both full-text and similarity search. Thanks to its cloud and cluster capabilities, Solr is well suited to handle big datasets and to apply distributed search and indexing which is highly relevant especially for long-term deployments.

In ScaR, different cores store information about items and interactions along with metadata about the generated recommendations. A typical configuration in ScaR uses three Solr cores. These are named “items”, “interactions” and “feedbacks”. The items core contains the items to recommend, the interactions core contains the recorded user interactions, and the feedbacks core contains the calculated recommendations as well as information regarding the evaluation of the system.

For deployment, the official Docker image is used, in a form that is adapted to the specifics of a project.

3.2 Deployment

ScaR is provided via Docker Images and runs in Docker Containers with the help of Docker Compose.

3.2.1 Docker Container

Docker[7] is a free software for isolating applications by using Operation System-level virtualization. Applications deployed with Docker are executed in entities called containers. Containers are standardized, lightweight, standalone, executable units. They contain the application with all its dependencies. These may be code, libraries, runtime, system tools, configurations and more. These containers are encapsulated environments and the delivered application runs within this environment. This isolation of containers from each other prevents conflicts between applications. Containers can be easily transported and installed as files, denoted as "images". This significantly simplifies the deployment of the ScaR modules. In comparison to virtual machines, Docker Containers are more efficient because with Docker the system hardware is not virtualized. Docker Images are also significantly smaller than images of virtual machines.

[5] https://solr.apache.org

[6] https://www.elastic.co/elasticsearch/

[7] https://www.docker.com

Containers are isolated from each other by definition. Following the Microservice architectural design pattern [7], every ScaR module is deployed in its own container as depicted in Figure 8. This approach requires and guarantees a certain level of modularization. With this high degree of modularization, we ensure flexibility and a high degree of maintainability of the code and of the deployed system.

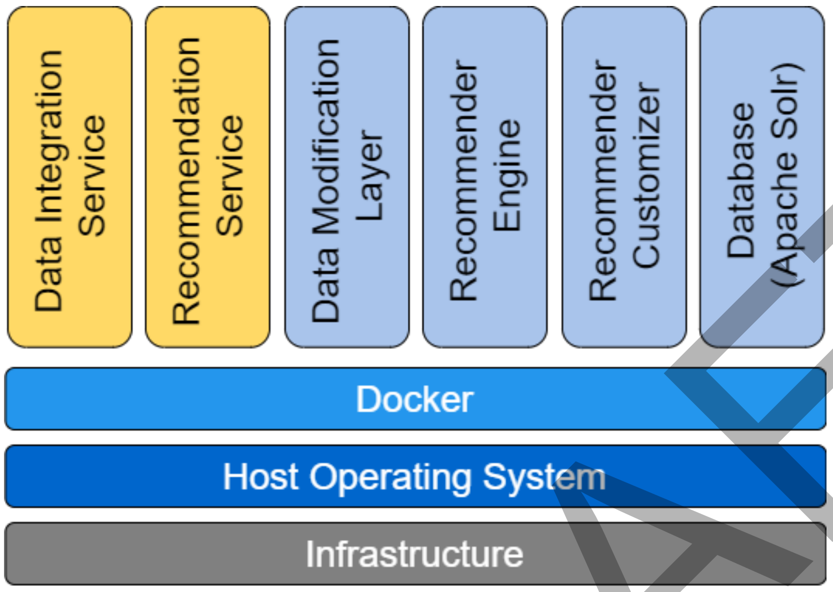


FIGURE 8: SCAR MODULES DEPLOYED AS DOCKER CONTAINERS.

4 INTEGRATION STRATEGIES OF THE SERVICE IN GOTRIPLE

4.1 Deployment Setting

To meet the current requirements of GoTriple, the recommendation services are provided via two separate and continuously deployed instances which offer a development and a production environment, respectively. To operate the recommendation services for two different maturity levels, Know-Center offers increased backup functionality for the production-ready service instance. Hence, to support the development cycle executed in Triple, most recent functionalities and bug-fixes can be accessed via the development server at https://triple.know-center.at/sp/swagger-ui.html. To make sure only stable versions of the service are used for the GoTriple production environment, the production-ready service can be accessed via https://prod-triple.know-center.at/sp/swagger-ui.html. Both instances are secured with the help of HTTP Basic Authentication requiring username and password to login on top of HTTP connections which are encrypted with TSL.

Continuous integration of the recommendation services is supported by the use of Jenkins hosted at Know-Center which in turn is also used to build and deploy the latest Docker Images of respective services to the two different environments. For ease of deployment and keeping the services independent of the underlying virtual machine environments, the recommendation services are run via Docker (more specifically via Docker Compose).

4.2 System Architecture

The system architecture is based on the scalable recommendation framework ScaR. Its general infrastructure and modules are depicted in Figure 9. The Data Integration Service and the Recommendation Service, which provide the REST interface for the GoTriple platform, are customized to meet Triple's needs. The Apache Solr cores are configured to store data and interactions from the Triple domain, and the Recommender Customizer is equipped with recommendation profiles for Triple's use cases.

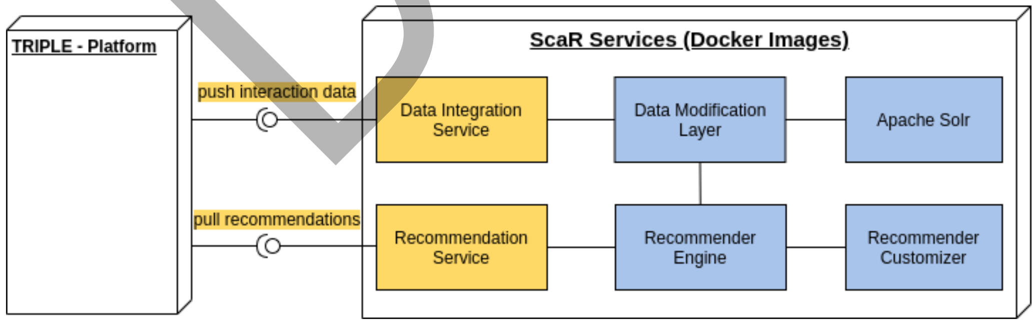


FIGURE 9: SYSTEM ARCHITECTURE

4.3 Recommender REST API

This section provides an overview of the REST API, which can be divided into Data Ingestion Services and Recommendation Services. The former includes resources for storing user interaction to the RS and a resource to trigger the data synchronization process. The latter provides the services for requesting recommendations from the system.

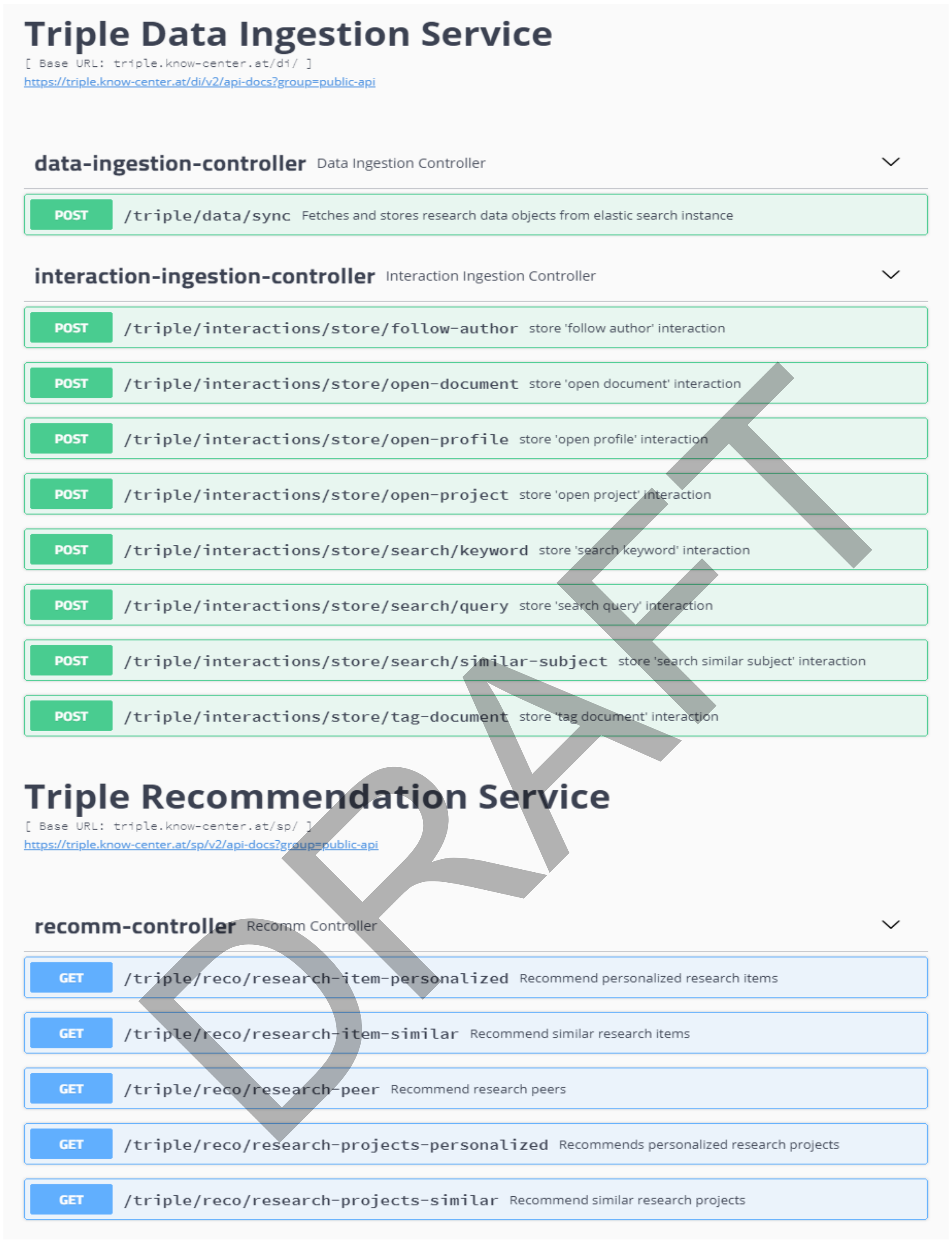
Please note that the schemas also reflect fields that are yet not implemented in GoTriple but have been agreed upon for future platform releases. This is true for instance for the search query objects which can only be partly completed by the current GoTriple GUI input, but already offer additional fields for a planned "advanced search" extension.

Both API parts are described in OpenAPI Specification Version 2.0 format, which is formerly known as "Swagger RESTful API Documentation Specification" [13, 14].

To developers the interface specification is presented through the visual documentation tool SpringFox [15] online [8] [9], as shown in Figure 10.

[8] On the development server the API specifications are located at: https://triple.know-center.at/sp/swagger-ui.html and https://triple.know-center.at/di/swagger-ui.html

[9] On the production server the API specifications are located at: https://prod-triple.know-center.at/sp/swagger-ui.html and https://prod-triple.know-center.at/di/swagger-ui.html


Triple Data Ingestion Service

[Base URL: triple.know-center.at/di/]

https://triple.know-center.at/di/v2/api-docs?group=public-api

data-ingestion-controller Data Ingestion Controller

POST /triple/data/sync Fetches and stores research data objects from elastic search instance

interaction-ingestion-controller Interaction Ingestion Controller

POST /triple/interactions/store/follow-author store 'follow author' interaction

POST /triple/interactions/store/open-document store 'open document' interaction

POST /triple/interactions/store/open-profile store 'open profile' interaction

POST /triple/interactions/store/open-project store 'open project' interaction

POST /triple/interactions/store/search/keyword store 'search keyword' interaction

POST /triple/interactions/store/search/query store 'search query' interaction

POST /triple/interactions/store/search/similar-subject store 'search similar subject' interaction

POST /triple/interactions/store/tag-document store 'tag document' interaction

Triple Recommendation Service

[Base URL: triple.know-center.at/sp/]

https://triple.know-center.at/sp/v2/api-docs?group=public-api

recomm-controller Recomm Controller

GET /triple/reco/research-item-personalized Recommend personalized research items

GET /triple/reco/research-item-similar Recommend similar research items

GET /triple/reco/research-peer Recommend research peers

GET /triple/reco/research-projects-personalized Recommends personalized research projects

GET /triple/reco/research-projects-similar Recommend similar research projects

FIGURE 10: AVAILABLE REST-SERVICES.

4.3.1 Triple Data Ingestion Service

The Data Ingestion Service resources are provided by the Data Integration Service module. These REST resources are responsible for storing interaction data in the RS and for triggering the data synchronization process. The resources provided are described in more detail below.

/triple/data/sync

A call to this resource triggers the data synchronization process, which fetches and stores research data objects from the GoTriple Elasticsearch instance and stores it in the RS database.

Type: POST

Parameters: None

Responses:

TABLE 1: API RESPONSES FOR /TRIPLE/DATA/SYNC

Code	Description	Schema
200	OK	GeneralResult

/triple/interactions/store/follow-author

This API call stores a "follow author" interaction to the backend database. The interaction is provided as a JSON object following the "Follow Author Interaction" schema. The response is a GeneralResult object.

Type: POST

Parameters:

TABLE 2: API PARAMETERS OF /TRIPLE/INTERACTIONS/STORE/FOLLOW-AUTHOR

Name	Located in	Description	Required	Schema
interaction	body	The follow author interaction data object.	Yes	Follow Author Interaction

Responses:

TABLE 3: API RESPONSES OF /TRIPLE/INTERACTIONS/STORE/FOLLOW-AUTHOR

Code	Description	Schema
200	OK	GeneralResult

/triple/interactions/store/open-document

This API call stores an "open document" interaction to the backend database. The interaction is provided as a JSON object following the "Open Document Interaction" schema.

Type: POST

Parameters:

TABLE 4: API PARAMETERS OF /TRIPLE/INTERACTIONS/STORE/OPEN-DOCUMENT

Name	Located in	Description	Required	Schema
interaction	body	The open document interaction data object.	Yes	Open Document Interaction

Responses:

TABLE 5: API RESPONSES OF /TRIPLE/INTERACTIONS/STORE/OPEN-DOCUMENT

Code	Description	Schema
200	OK	GeneralResult

/triple/interactions/store/open-profile

This API call stores an "open profile" interaction to the backend database. The interaction is provided as a JSON object following the "Open Profile Interaction" schema. The response is a GeneralResult object.

Type: POST

Parameters:

TABLE 6: API PARAMETERS OF /TRIPLE/INTERACTIONS/STORE/OPEN-PROFILE

Name	Located in	Description	Required	Schema
interaction	body	The open profile interaction data object.	Yes	Open Profile Interaction

Responses:

TABLE 7: API RESPONSES OF /TRIPLE/INTERACTIONS/STORE/OPEN-PROFILE

Code	Description	Schema
200	OK	GeneralResult

/triple/interactions/store/open-project

This API call stores an "open project" interaction to the backend database. The interaction is provided as a JSON object following the "Open Project Interaction" schema. The response is a GeneralResult object.

Type: POST

Parameters:

TABLE 8: API PARAMETERS OF /TRIPLE/INTERACTIONS/STORE/OPEN-PROJECT

Name	Located in	Description	Required	Schema
interaction	body	The open project interaction data object.	Yes	Open Project Interaction

Responses:

TABLE 9: API RESPONSES OF /TRIPLE/INTERACTIONS/STORE/OPEN-PROJECT

Code	Description	Schema
200	OK	GeneralResult

/triple/interactions/store/search/keyword

This API call stores "search keyword" interaction to the backend database. The interaction is provided as a JSON object following the "Search Keyword Interaction" schema. The response is a GeneralResult object.

Type: POST

Parameters:

TABLE 10: API PARAMETERS OF /TRIPLE/INTERACTIONS/STORE/SEARCH/KEYWORD

Name	Located in	Description	Required	Schema
interaction	body	The search keyword interaction data object.	Yes	Search Keyword Interaction

Responses:

TABLE 11: API RESPONSES OF /TRIPLE/INTERACTIONS/STORE/SEARCH/KEYWORD

Code	Description	Schema
200	OK	GeneralResult

/triple/interactions/store/search/query

This API call stores a "search query" interaction to the backend database. The interaction is provided as a JSON object following the "Search Query Interaction" schema. The response is a GeneralResult object.

Type: POST

Parameters:

TABLE 12: API PARAMETERS OF /TRIPLE/INTERACTIONS/STORE/SEARCH/QUERY

Name	Located in	Description	Required	Schema
interaction	body	The search query interaction data object.	Yes	Search Query Interaction

Responses:

TABLE 13: API RESPONSES OF /TRIPLE/INTERACTIONS/STORE/SEARCH/QUERY

Code	Description	Schema
200	OK	GeneralResult

/triple/interactions/store/search/similar-subject

This API call stores a "search similar subject" interaction to the backend database. The interaction is provided as a JSON object following the "Search Similar Subject Interaction" schema. The response is a GeneralResult object.

Type: POST

Parameters:

TABLE 14: API PARAMETERS OF /TRIPLE/INTERACTIONS/STORE/SEARCH/SIMILAR-SUBJECT

Name	Located in	Description	Required	Schema
interaction	body	The search similar interaction data object.	Yes	Search Similar Subject Interaction

Responses:

TABLE 15: API RESPONSES OF /TRIPLE/INTERACTIONS/STORE/SEARCH/SIMILAR-SUBJECT

Code	Description	Schema
200	OK	GeneralResult

/triple/interactions/store/tag-document

Search Similar Subject Interaction a "tag document" interaction to the backend database. The interaction is provided as a JSON object. The response is a GeneralResult object.

Type: POST

Parameters:

TABLE 16: API PARAMETERS OF /TRIPLE/INTERACTIONS/STORE/TAG-DOCUMENT

Name	Located in	Description	Required	Schema
interaction	body	The tag document interaction data object.	Yes	Tag Document Interaction

Responses:

TABLE 17: API RESPONSES OF /TRIPLE/INTERACTIONS/STORE/TAG-DOCUMENT

Code	Description	Schema
200	OK	GeneralResult

4.3.2 Triple Recommendation Service

The recommendation service resources are provided by the Recommendation Service module. These REST resources are used to retrieve recommendations from the RS. The provided resources are described in detail below. The TripleRecoResult is described in Section 4.3.3 with more detail.

/triple/reco/research-item-personalized

An API call to retrieve personalized research items as recommendations. Parameters are the number of recommendations to generate and the ID of the user for whom the recommendations should be generated. The response contains the list of recommended item IDs and optionally some additional meta data.

Type: GET

Parameters:

TABLE 18: API PARAMETERS OF /TRIPLE/RECO/RESEARCH-ITEM-PERSONALIZED

Name	Located in	Description	Required	Schema
count	query	Number of recommendations to be generated.	No	integer
userId	query	The ID of the user for whom the recommendations should be generated.	Yes	string

Responses:

TABLE 19: API RESPONSES OF /TRIPLE/RECO/RESEARCH-ITEM-PERSONALIZED

Code	Description	Schema
200	OK	TripleRecoResult

/triple/reco/research-item-similar

An API call to retrieve similar research items as recommendations. Parameters are the number of recommendations to generate, the ID of the research item which serves as an example for the recommendations, and the ID of the user for whom the recommendations should be generated. The response contains the list of recommended item IDs and optionally some additional meta data.

Type: GET

Parameters:

TABLE 20: API PARAMETERS OF /TRIPLE/RECO/RESEARCH-ITEM-SIMILAR

Name	Located in	Description	Required	Schema
count	query	Number of recommendations to be generated.	No	integer
researchItemId	query	The ID of the research item which serves as an example for the recommendations.	Yes	string
userId	query	The ID of the user for whom the recommendations should be generated.	Yes	string

Responses:

TABLE 21: API RESPONSES OF /TRIPLE/RECO/RESEARCH-ITEM-SIMILAR

Code	Description	Schema
200	OK	TripleRecoResult

/triple/reco/research-peer

An API call to retrieve research peers as recommendations. Parameters are the number of recommendations to generate the ID of the research peer which serves as an example for the recommendations, and the ID of the user for whom the recommendations should be generated. The response contains the list of recommended user IDs and optionally some additional meta data.

Type: GET

Parameters:

TABLE 22: API PARAMETERS OF /TRIPLE/RECO/RESEARCH-PEER

Name	Located in	Description	Required	Schema
count	query	Number of recommendations to be generated.	No	integer
researchPeerId	query	The ID of the research peer which serves as an example for the recommendations.	Yes	string
userId	query	The ID of the user for whom the recommendations should be generated.	Yes	string

Responses:

TABLE 23: API RESPONSE OF /TRIPLE/RECO/RESEARCH-PEER

Code	Description	Schema
200	OK	TripleRecoResult

/triple/reco/research-projects-personalized

An API call to retrieve personalized research projects as recommendations. Parameters are the number of recommendations to generate and the ID of the user for whom the recommendations should be generated. The response contains the list of recommended item IDs and optionally some additional meta data.

Type: GET

Parameters:

TABLE 24: API PARAMETERS OF /TRIPLE/RECO/RESEARCH-PROJECTS-PERSONALIZED

Name	Located in	Description	Required	Schema
count	query	Number of recommendations to be generated.	No	integer
userId	query	The ID of the user for whom the recommendations should be generated.	Yes	string

Responses:

TABLE 25: API RESPONSE OF /TRIPLE/RECO/RESEARCH-PROJECTS-PERSONALIZED

Code	Description	Schema
200	OK	TripleRecoResult

/triple/reco/research-projects-similar

An API call to retrieve similar research projects as recommendations. Parameters are the number of recommendations to generate, the ID of the research project which serves as an example for the recommendations, and the ID of the user for whom the recommendations should be generated. The response contains the list of recommended item IDs and optionally some additional meta data.

Type: GET

Parameters:

TABLE 26: API PARAMETERS OF /TRIPLE/RECO/RESEARCH-PROJECTS-SIMILAR

Name	Located in	Description	Required	Schema
count	query	Number of recommendations to be	No	integer

		generated.		
researchProjectId	query	The ID of the research project which serves as an example for the recommendations.	Yes	string
userId	query	The ID of the user for whom the recommendations should be generated.	Yes	string

Responses:

TABLE 27: API RESPONSE OF /TRIPLE/RECO/RESEARCH-PROJECTS-SIMILAR

Code	Description	Schema
200	OK	TripleRecoResult

4.3.3 API Data Schema

On several occasions JSON data objects are used in the API as parameters or response objects. These objects must be in line with the schemas described in this section but do not need to provide for all of their fields.

Author

The Author data object is used as an attribute of the Follow Author Interaction data object. It represents an author identified by the author ID and optionally the author's name.

TABLE 28: DATA SCHEMA "AUTHOR"

Name	Type	Description	Required
Id	string	The unique author ID	Yes
Name	string	The author's name	No

Follow Author Interaction

The Follow Author Interaction data object is used to store a follow author interaction to the database by use of the Data Integration Module's REST API.

TABLE 29: DATA SCHEMA "FOLLOW AUTHOR INTERACTION"

Name	Type	Description	Required
Author	Author	Author data object	Yes
recommenderId	string	Optional reference of the recommended item.	No
sessionId	string	The user's session ID	No
timestamp	long	Timestamp of the interaction in milliseconds, e.g., May 15, 2020 = 1589500800000	Yes
userId	string	The user's user ID	No

GeneralResult

The GeneralResult data object solves as result object for use cases where no data needs to be returned to the caller. It contains the HTTP status code and optionally a human readable message.

TABLE 30: DATA SCHEMA "GENERALRESULT"

Name	Type	Description	Required
http_status	integer	HTTP status code	Yes
message	string	Human readable status message.	No

Open Document Interaction

The Open Document Interaction data object is used to store an open document interaction to the database by use of the Data Integration Module's REST API.

TABLE 31: DATA SCHEMA "OPEN DOCUMENT INTERACTION"

Name	Type	Description	Required
documentId	string	ID of the opened document	Yes
openDocumentInteractionType	string	Type of the open document interaction. Valid values are: • OPEN_DESCRIPTION • ACCESS_DOCUMENT • EXPORT_INFORMATION	Yes
recommenderId	string	Optional reference on the recommended item.	No
sessionId	string	The user's session ID	No
timestamp	long	Timestamp of the interaction in milliseconds, e.g., May 15, 2020 = 1589500800000	Yes
userId	string	The user's user ID	No

Open Profile Interaction

The Open Profile Interaction data object is used to store an open profile interaction to the database by use of the Data Integration Module's REST API.

TABLE 32: DATA SCHEMA "OPEN PROFILE INTERACTION"

Name	Type	Description	Required
profileId	string	ID of the opened profile.	Yes
openProfileInteractionType	string	Type of the open profile interaction. Valid values are:	Yes

		• OPEN_PROFILE_DESCRIPTION • ACCESS_PROFILE • EXPORT_PROFILE_INFORMATION	
recommenderId	string	Optional reference on the recommended item.	No
sessionId	string	The user's session ID	No
timestamp	long	Timestamp of the interaction in milliseconds, e.g., May 15, 2020 = 1589500800000	Yes
userId	string	The user's user ID	No

Open Project Interaction

The Open Project Interaction data object is used to store an open project interaction to the database by use of the Data Integration Module's REST API.

TABLE 33: DATA SCHEMA "OPEN PROJECT INTERACTION"

Name	Type	Description	Required
projectId	string	ID of the opened project	Yes
openProfileInteractionType	string	Type of the open project interaction. Valid values are: • OPEN_PROJECT_DESCRIPTION • ACCESS_PROJECT • EXPORT_PROJECT_INFORMATION	Yes
recommenderId	string	Optional reference on the recommended item.	No
sessionId	string	The user's session ID	No
timestamp	long	Timestamp of the interaction in milliseconds, e.g., May 15, 2020 = 1589500800000	Yes
userId	string	The user's user ID	No

Search Keyword Interaction

The Search Keyword Interaction data object is used to store a search keyword interaction to the database by use of the Data Integration Module's REST API.

TABLE 34: DATA SCHEMA "SEARCH KEYWORD INTERACTION"

Name	Type	Description	Required
keyword	string	Query keyword	Yes
recommenderId	string	Optional reference on the recommended item.	No
sessionId	string	The user's session ID	No
timestamp	long	Timestamp of the interaction in milliseconds, e.g., May 15, 2020 = 1589500800000	Yes

userId	string	The user's user ID	No

Search Query Interaction

The Search Query Interaction data object is used to store a search query interaction to the database by use of the Data Integration Module's REST API.

TABLE 35: DATA SCHEMA "SEARCH QUERY INTERACTION"

Name	Type	Description	Required
query	Search-Query-Interaction Query	Query data object. See: "Search-Query-Interaction Query"	Yes
recommenderId	string	Optional reference on the recommended item.	No
sessionId	string	The user's session ID	No
timestamp	long	Timestamp of the interaction in milliseconds, e.g., May 15, 2020 = 1589500800000	Yes
userId	string	The user's user ID	No

Search Similar Subject Interaction

The Search Similar Subject Interaction data object is used to store a search similar subject interaction to the database by use of the Data Integration Module's REST API.

TABLE 36: DATA SCHEMA "SEARCH SIMILAR SUBJECT INTERACTION"

Name	Type	Description	Required
subject	string	Subject	Yes
language	string	Language	No
recommenderId	string	Optional reference on the recommended item.	No
sessionId	string	The user's session ID	No
timestamp	long	Timestamp of the interaction in milliseconds, e.g., May 15, 2020 = 1589500800000	Yes
userId	string	The user's user ID	No

Search-Query-Interaction Query

The Search-Query-Interaction Query data object is used as an attribute of a Search Query Interaction data object, where it represents the search query.

TABLE 37: DATA SCHEMA "SEARCH-QUERY-INTERACTION QUERY"

Name	Type	Description	Required
discipline	string list	List of disciplines	No

text	string	Query text	No
type	string list	List of types	No

Tag Document Interaction

The Tag Document Interaction data object is used to store a tag document interaction to the database by use of the Data Integration Module's REST API.

TABLE 38: DATA SCHEMA "TAG DOCUMENT INTERACTION"

Name	Type	Description	Required
documentId	string	ID of the opened document	Yes
userTags	string list	List of tags	Yes
recommenderId	string	Optional reference on the recommended item.	No
sessionId	string	The user's session ID	No
timestamp	long	Timestamp of the interaction in milliseconds, e.g., May 15, 2020 = 1589500800000	Yes
userId	string	The user's user ID	No

TripleRecoResult

The TripleRecoResult data object is used in the response of recommendation request calls, where it represents the generated recommendations.

TABLE 39: DATA SCHEMA "TRIPLERECORESULT"

Name	Type	Description	Required
http_status	integer	HTTP status message.	Yes
message	string	Human readable message.	No
reco_id	string	Recommendation Id.	No
results	string list	Recommendation results.	No

4.4 Recommendation Service

The modular design of ScaR allows to follow a standardized recommendation sequence which is depicted in Figure 4. Integration efforts for the Recommendation Service focussed on the adaption of the REST interface provided to GoTriple and the associated validation routines. The REST interface and the validation routines are now tailored to TRIPLE's use cases. Valid recommendation requests are parsed and mapped to the associated recommendation profile ID. Then the Recommendation Service requests the recommendations from the Recommender Engine and delivers the retrieved results to the caller. The adaptation to the domain data is done mainly via recommendation profiles which are described in the next section.

4.5 Recommendation Profiles

For the integration with GoTriple two different types of algorithms are applied so far:

- Collaborative Filtering and
- Content Based Filtering

To instantiate these algorithms for the recommendation of research items, two instances of recommender profiles have been created:

- triple_document_cf.yaml: Profile for recommending personalized research items.
- triple_document_item_cb.yaml: Profile for recommending similar research items.

After respective data is available, additional profiles for the aforementioned algorithms will be prepared and ~~will be~~ set up for the recommendation of research projects and research peers:

- triple_project_cf.yaml: Profile for recommending personalized research projects.
- triple_project_item_cb.yaml: Profile for recommending similar research projects.
- triple_user_cf.yaml: Profile for recommending individually matched research peers.

4.5.1 Personalized research items

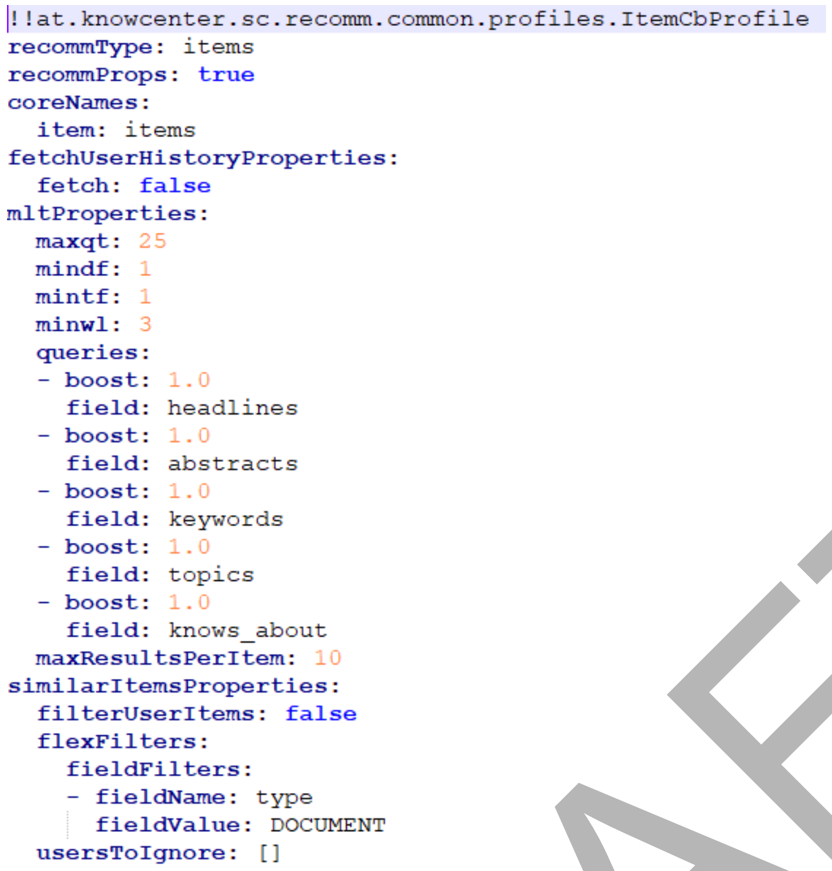
The recommender profile for recommending personalized research items is configured in the file triple_document_cf.yaml whose contents are shown in Figure 11. It uses the InteractionCfProfile which is a profile for the Collaborative Filtering algorithm. It is configured to recommend entities from the items core which is named "items". Up to 10 items from the user's history are considered. These items are extracted from the "document_id" field of the user's last 40 interactions. Users are identified either by their user ID or their session ID. The interactions taken into account are of the types "TAG_DOCUMENT", "OPEN_DESCRIPTION", "ACCESS_DOCUMENT", and "EXPORT_INFORMATION". Up to 40 similar users are considered for generating recommendations. Users must have at least one common item to be considered for similarity calculations. Similar items are considered when they are of type "DOCUMENT". Item similarity calculations are based on the interaction fields "users_tag_document", "users_access_document", "users_export_information", and "users_open_description".

```
!!at.knowcenter.sc.recomm.common.profiles.InteractionCfProfile
recommType: items
coreNames:
  item: items
recommProps: true
fetchUserHistoryProperties:
  fetch: true
  itemCount: 10
  interactionCoreProperties:
    interactionCount: 40
    timeFieldName: timestamp
    userFieldNames:
      - user_id
      - session_id
    itemFieldName: document_id
    interactionTypes:
      - TAG_DOCUMENT
      - OPEN_DESCRIPTION
      - ACCESS_DOCUMENT
      - EXPORT_INFORMATION
similarUsersProperties:
  similarUserCount: 40
  topKSimilarUserCount: 40
  minItemOverlap: 1
similarItemsProperties:
  filterUserItems: true
  flexFilters:
    fieldFilters:
    - fieldName: type
      fieldValue: DOCUMENT
  interactionFields:
    fields:
    - name: users_tag_document
    - name: users_access_document
    - name: users_export_information
    - name: users_open_description
  usersToIgnore: []
```

FIGURE 11: TRIPLE_DOCUMENT_CF.YAML

4.5.2 Similar Research Items

The recommender profile for recommending similar research items is configured in the file triple_document_item_cb.yaml whose contents are shown in Figure 12. It uses the ItemCbProfile which is a profile for the Content Based Filtering algorithm. It is configured to recommend entities from the items core which is named “items”. User history is not considered for recommendations. Similar items are identified based on the contents of the following fields using a TF-IDF score [16] for: “abstracts”, “headlines”, “keywords”, “topics”, “knows_about”, , and. Currently, all fields are considered with the same importance. Words shorter than three characters are ignored and there are no limitations on term frequency and document frequency. The length of the generated similarity query is limited to the 25 most important terms and the query result is limited to 10 items. Similar items are considered for recommendation when they are of type “DOCUMENT”.



```
!!at.knowcenter.sc.recomm.common.profiles.ItemCbProfile
recommType: items
recommProps: true
coreNames:
  item: items
fetchUserHistoryProperties:
  fetch: false
mltProperties:
  maxqt: 25
  mindf: 1
  mintf: 1
  minwl: 3
  queries:
  - boost: 1.0
    field: headlines
  - boost: 1.0
    field: abstracts
  - boost: 1.0
    field: keywords
  - boost: 1.0
    field: topics
  - boost: 1.0
    field: knows_about
  maxResultsPerItem: 10
similarItemsProperties:
  filterUserItems: false
  flexFilters:
    fieldFilters:
    - fieldName: type
      fieldValue: DOCUMENT
  usersToIgnore: []
```

FIGURE 12: TRIPLE_DOCUMENT_ITEM_CB.YAML

4.6 Data Ingestion Service

The Data Ingestion Service provides the REST interface for ingesting data and user interactions. Its interface and functionality have been customized for use with the GoTriple platform. In terms of data integration, we need to distinguish two types of data: i) static and ii) dynamic data. As static data we consider data that describe items relevant to the recommendation process, e.g., metadata of documents, projects and researchers that do not change frequently. The term dynamic data is used for user interaction data that is produced and updated at the time of occurrence.

4.6.1 Integration of Static Data

TRIPLE's research data objects serve as the items in the RS. Meta-data describing these research data objects are retrieved from the Elasticsearch instance of GoTriple which for now happens whenever a synchronisation request is made, while a subsequent version will implement a periodic automatic synchronization e.g., once a day. Retrieved data objects are processed and sent to the Data Modification Layer for storage in the database. The GoTriple data ingestion process is visualized in Figure 13.

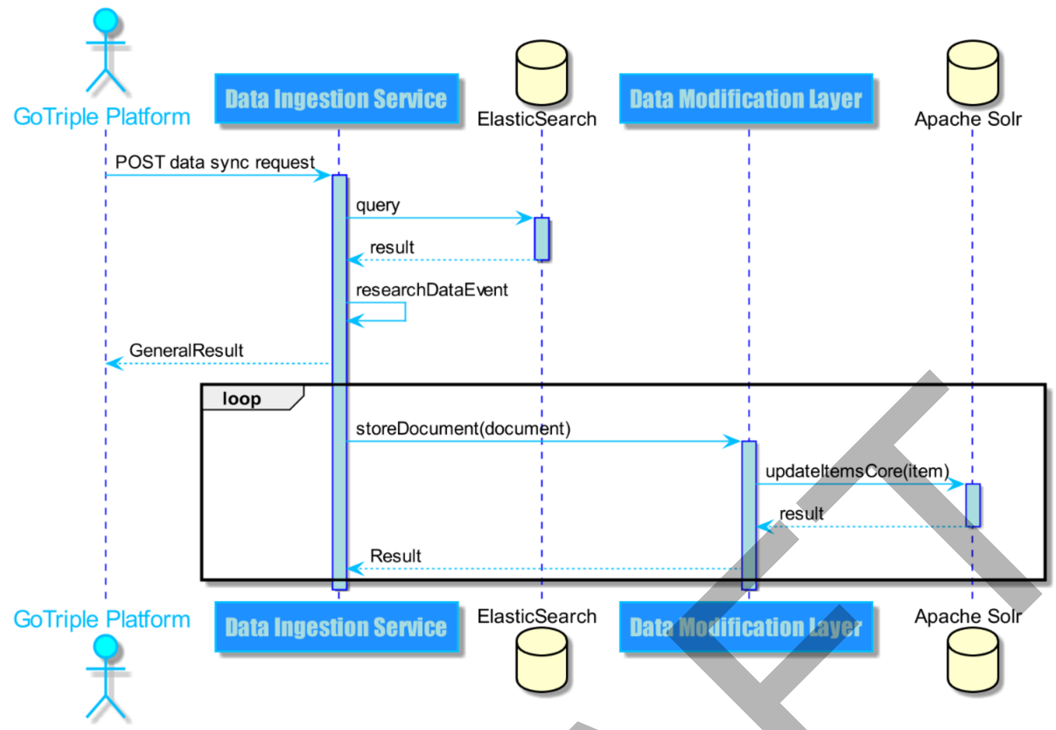


FIGURE 13: INGESTION OF RESEARCH DATA OBJECTS IN GOTRIPLE.

4.6.2 Integration of Dynamic Data

The approach for ingesting GoTriple's user interaction data is depicted in Figure 14. Interactions are posted to the Data Ingestion Services' REST interface, and subsequently processed and sent to the Data Modification Layer for storing. Interaction data validation routines have been adapted to accurately reflect the specifics of GoTriple's interaction data. The processing routines have also been adjusted so that TRIPLE-specific attributes are handled correctly and result in the appropriate updates to the backend database.

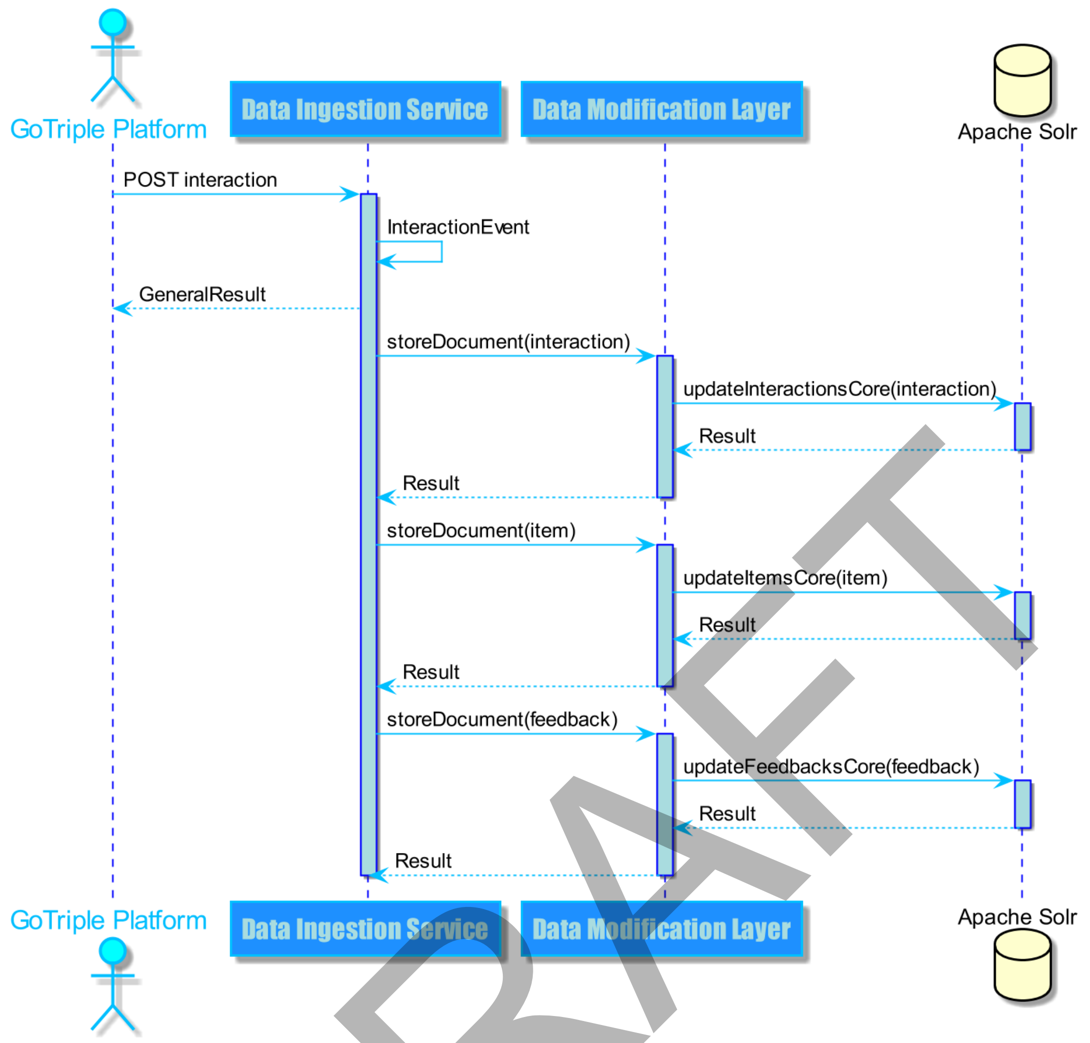


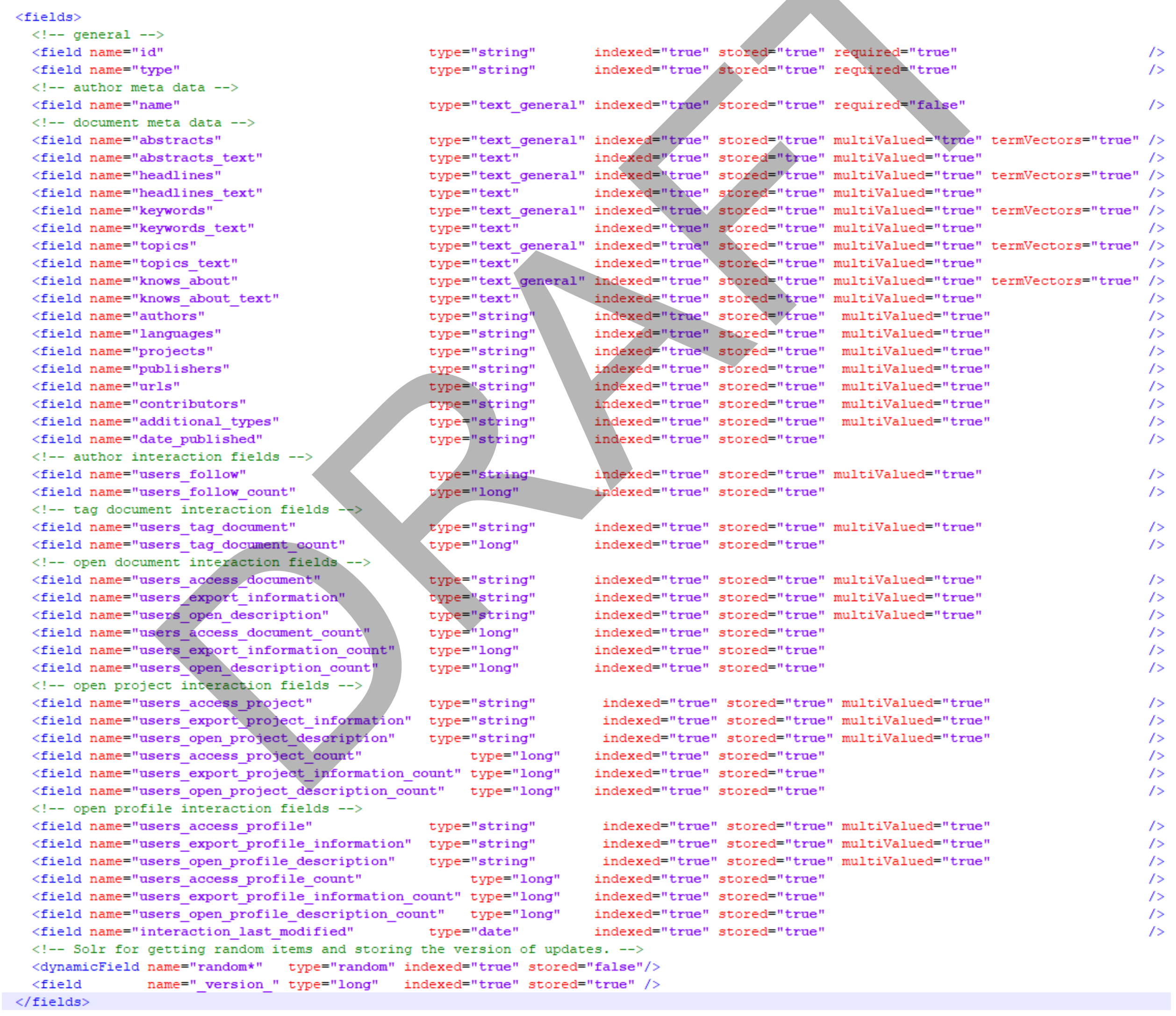
FIGURE 14: INGESTION OF USER INTERACTION DATA IN GOTRIPLE.

4.7 Apache Solr

In GoTriple three Solr cores are used. These are named “items”, “interactions” and “feedbacks”. The *Items core* contains the documents, the *Interactions core* contains the recorded interactions, and the *Feedbacks core* contains the calculated recommendations as well as information regarding the evaluation of the system. Please note that the core schema also reflects fields that are yet not implemented in GoTriple but have been agreed upon for future platform releases.

4.7.1 The Items Core

Item documents (e.g., research documents, projects, peers) are stored in the Items core. In a typical setting these are the objects which can be recommended to the users. From a conceptual viewpoint the fields of an item document serve one of two purposes. The first purpose is to describe and identify the item itself. These are fields which hold meta-data like item ID, type, keywords and item descriptions or in case of textual documents the item’s content. The contents of these fields are static. They are defined when the item is first stored to the backend database and do not change unless an update is explicitly requested. The second purpose is to capture how users interact with the item. These fields contain for example user IDs and interaction counts. They represent who interacted with an item and how many interactions an item gets. These field contents are updated every time an interaction with the specific item is logged. For this, the atomic update capability of Solr is used, which allows for the isolated updating of fields in a document. The Items core's configuration was adapted so that GoTriple items can be stored in the Items core. This domain specific configuration is located in the fields section of the schema.xml configuration file corresponding to the Items core. Figure 15 shows the configured fields and their data types as provided in the core's schema configuration.



```
<fields>
  <!-- general -->
  <field name="id"                                type="string"       indexed="true" stored="true" required="true"                       />
  <field name="type"                              type="string"       indexed="true" stored="true" required="true"                       />
  <!-- author meta data -->
  <field name="name"                              type="text_general" indexed="true" stored="true" required="false"                      />
  <!-- document meta data -->
  <field name="abstracts"                         type="text_general" indexed="true" stored="true" multiValued="true" termVectors="true" />
  <field name="abstracts_text"                    type="text"         indexed="true" stored="true" multiValued="true"                    />
  <field name="headlines"                         type="text_general" indexed="true" stored="true" multiValued="true" termVectors="true" />
  <field name="headlines_text"                    type="text"         indexed="true" stored="true" multiValued="true"                    />
  <field name="keywords"                          type="text_general" indexed="true" stored="true" multiValued="true" termVectors="true" />
  <field name="keywords_text"                     type="text"         indexed="true" stored="true" multiValued="true"                    />
  <field name="topics"                            type="text_general" indexed="true" stored="true" multiValued="true" termVectors="true" />
  <field name="topics_text"                       type="text"         indexed="true" stored="true" multiValued="true"                    />
  <field name="knows_about"                       type="text_general" indexed="true" stored="true" multiValued="true" termVectors="true" />
  <field name="knows_about_text"                  type="text"         indexed="true" stored="true" multiValued="true"                    />
  <field name="authors"                           type="string"       indexed="true" stored="true"  multiValued="true"                   />
  <field name="languages"                         type="string"       indexed="true" stored="true"  multiValued="true"                   />
  <field name="projects"                          type="string"       indexed="true" stored="true"  multiValued="true"                   />
  <field name="publishers"                        type="string"       indexed="true" stored="true"  multiValued="true"                   />
  <field name="urls"                              type="string"       indexed="true" stored="true"  multiValued="true"                   />
  <field name="contributors"                      type="string"       indexed="true" stored="true"  multiValued="true"                   />
  <field name="additional_types"                  type="string"       indexed="true" stored="true"  multiValued="true"                   />
  <field name="date_published"                    type="string"       indexed="true" stored="true"                                       />
  <!-- author interaction fields -->
  <field name="users_follow"                      type="string"       indexed="true" stored="true" multiValued="true"                    />
  <field name="users_follow_count"                type="long"         indexed="true" stored="true"                                       />
  <!-- tag document interaction fields -->
  <field name="users_tag_document"                type="string"       indexed="true" stored="true" multiValued="true"                    />
  <field name="users_tag_document_count"          type="long"         indexed="true" stored="true"                                       />
  <!-- open document interaction fields -->
  <field name="users_access_document"             type="string"       indexed="true" stored="true" multiValued="true"                    />
  <field name="users_export_information"          type="string"       indexed="true" stored="true" multiValued="true"                    />
  <field name="users_open_description"            type="string"       indexed="true" stored="true" multiValued="true"                    />
  <field name="users_access_document_count"       type="long"         indexed="true" stored="true"                                       />
  <field name="users_export_information_count"    type="long"         indexed="true" stored="true"                                       />
  <field name="users_open_description_count"      type="long"         indexed="true" stored="true"                                       />
  <!-- open project interaction fields -->
  <field name="users_access_project"              type="string"        indexed="true" stored="true" multiValued="true"                   />
  <field name="users_export_project_information"  type="string"        indexed="true" stored="true" multiValued="true"                   />
  <field name="users_open_project_description"    type="string"        indexed="true" stored="true" multiValued="true"                   />
  <field name="users_access_project_count"             type="long"    indexed="true" stored="true"                                       />
  <field name="users_export_project_information_count" type="long"    indexed="true" stored="true"                                       />
  <field name="users_open_project_description_count"   type="long"    indexed="true" stored="true"                                       />
  <!-- open profile interaction fields -->
  <field name="users_access_profile"              type="string"        indexed="true" stored="true" multiValued="true"                   />
  <field name="users_export_profile_information"  type="string"        indexed="true" stored="true" multiValued="true"                   />
  <field name="users_open_profile_description"    type="string"        indexed="true" stored="true" multiValued="true"                   />
  <field name="users_access_profile_count"             type="long"    indexed="true" stored="true"                                       />
  <field name="users_export_profile_information_count" type="long"    indexed="true" stored="true"                                       />
  <field name="users_open_profile_description_count"   type="long"    indexed="true" stored="true"                                       />
  <field name="interaction_last_modified"         type="date"         indexed="true" stored="true"                                       />
  <!-- Solr for getting random items and storing the version of updates. -->
  <dynamicField name="random*"   type="random" indexed="true" stored="false"/>
  <field        name="_version_" type="long"   indexed="true" stored="true" />
</fields>
```

FIGURE 15: ITEMS CORE SCHEMA

For better comprehension, the fields can be divided into several property categories. This is not depicted in the database but added for a better understanding of related fields. The “general” category contains the item id and its type.

The fields in the “general” category hold information about the item itself. The “interaction data” fields contain information about the interactions and the list of session IDs associated with these interactions. The fields of the Items core are listed in Table 40.

The fields in the interaction data category are updated every time an interaction with the particular item is logged. For this, the Data Modification Layer uses the atomic update capability of Solr. This feature allows for updating fields of a document separately. The fields in the other categories reflect static item properties which are not affected by user interactions.

TABLE 40: ITEMS CORE FIELDS

Property category	Field name	Content
general	id	The ID of the item.
	type	Type of the stored item.
	abstracts	Tokenized abstracts of the document
	abstracts_text	Abstracts of the document.
	headlines	The tokenized “headline” property retrieved from the data source.
	headlines_text	The tokenized “headline” property retrieved from the data source.
	keywords	The tokenized “keyword” property retrieved from the data source.
	keywords_text	The “keyword” property retrieved from the data source.
	topics	The tokenized “topics” property retrieved from the data source.
	topics_text	The “topics” property retrieved from the data source.
	knows_about	The tokenized “knows_about” property retrieved from the data source.
	knows_about_text	The “knows_about” property retrieved from the data source.
	authors	Full names of the authors.
	languages	Languages in which the item is available.
	projects	List of project IDs associated with the item.
	publishers	List of publishers.
	urls	List of URLs.
	contributors	The “contributors” property retrieved

		from the data source.
	additional_types	The “additional_types” property retrieved from the data source.
	date_published	The “date_published” property retrieved from the data source.
interaction data	users_tag_document	An array of unique IDs pointing to users who tagged the document.
	users_tag_document_count	Total number of times the document was tagged.
	users_access_document	An array of unique IDs pointing to users who accessed the document.
	users_export_information	An array of unique IDs pointing to users who exported the document information.
	users_open_description	An array of unique IDs pointing to users who opened the document description.
	users_access_document_count	Total number of times the document was accessed.
	users_export_information_count	Total number of times the document information was exported.
	users_open_description_count	Total number of times the document description was opened.
	users_access_project	An array of unique IDs pointing to users who accessed the project.
	users_export_project_information	An array of unique IDs pointing to users who exported the project information.
	users_open_project_description	An array of unique IDs pointing to users who opened the project description.
	users_access_project_count	Total number of times the project was accessed.
	users_export_project_information_count	Total number of times the project information was exported.
	users_open_project_description_count	Total number of times the project description was opened.
	users_access_profile	An array of unique IDs pointing to users who accessed the profile.
	users_export_profile_information	An array of unique IDs pointing to users who exported the profile information.

	users_open_profile_description	An array of unique IDs pointing to users who opened the profile description.
	users_access_profile_count	Total number of times the profile was accessed.
	users_export_profile_information_count	Total number of times the profile information was exported.
	users_open_profile_description_count	Total number of times the profile description was opened.

4.7.2 The Interactions Core

The Interactions core contains interaction data of the users. Some fields are common for all interaction types, such as the interaction ID, the type and the timestamp. These fields form the minimal required information to create a valid interaction entry. Further commonly used fields are the user-ID and the item-ID that describe objects involved in the interaction. Additional fields can be added based on the project requirements. There are also fields that are only meaningful for a particular interaction type. They are meant to capture interaction metadata like for example the query settings in case of a search interaction or keywords in case of a tagging interaction. These fields depend on the domain properties and are defined during ScaR integration. Like the items core, the Interactions core was adapted to store interaction data of GoTriple users. This domain specific configuration is located in the fields section of the schema.xml configuration file corresponding to the interactions core. Figure 16 shows the configured fields and their data types as provided in the core's schema configuration.

```
<fields>

  <field name="id"          type="string" indexed="true" stored="true" required="true" />
  <field name="type"        type="string" indexed="true" stored="true" required="true" />
  <field name="timestamp"   type="date"   indexed="true" stored="true" required="true" />
  <field name="user_id"     type="string" indexed="true" stored="true" />
  <field name="session_id"  type="string" indexed="true" stored="true" />
  <field name="reco_id"     type="string" indexed="true" stored="true" />
  <field name="document_id" type="string" indexed="true" stored="true" />
  <field name="project_id"  type="string" indexed="true" stored="true" />
  <field name="author_id"   type="string" indexed="true" stored="true" />
  <field name="profile_id"  type="string" indexed="true" stored="true" />

  <!-- search query interaction fields -->
  <field name="disciplines" type="string" indexed="true" stored="true" multiValued="true" />
  <field name="query_text"  type="string" indexed="true" stored="true" />
  <field name="types"       type="string" indexed="true" stored="true" multiValued="true" />

  <!-- search similar subject interaction fields -->
  <field name="subject"     type="string" indexed="true" stored="true" />
  <field name="language"    type="string" indexed="true" stored="true" />

  <!-- search keyword interaction fields -->
  <field name="keyword"     type="string" indexed="true" stored="true" />

  <!-- tag document interaction fields -->
  <field name="user_tags"   type="string" indexed="true" stored="true" multiValued="true" />

  <!-- Solr for getting random items and storing the version of updates. -->
  <dynamicField name="random*"   type="random" indexed="true" stored="false"/>
  <field        name="_version_" type="long"   indexed="true" stored="true" />

</fields>
```

FIGURE 16: INTERACTIONS CORE SCHEMA

The fields can be divided into several property categories. The categories are not depicted in the database but added for a better understanding of related fields. The "general" category contains the fields, common to more than one interaction type. At least the interaction ID, type, and timestamp must be present for a valid entry. The other categories - "search query", "search similar subject", "search keyword", and "tag document" - are only used for one particular interaction type. The fields of the Interactions core are listed in

Table 41.

TABLE 41: INTERACTIONS CORE – FIELD DESCRIPTION

Property category	Field name	Content
general	id	The ID of the interaction.
	type	The type of the interaction.
	timestamp	The timestamp indicates at which point in time a certain interaction was made.
	user_id	The ID of the user who is responsible for the interaction.

	session_id	The session ID of the user who is responsible for the interaction.
	reco_id	The ID of the recommendation which resulted in the current interaction.
	document_id	The ID of the document the user interacted with.
	project_id	The ID of the project the user interacted with.
	author_id	The ID of the author the user interacted with.
	profile_id	The ID of the profile the user interacted with.
search query	disciplines	The list of disciplines stated in the query.
	query_text	The query text.
	types	The list of types stated in the query.
search similar subject	subject	The subject stated in the query.
	language	The language stated in the query.
search keyword	keyword	The keyword stated in the query.
tag document	user_tags	The list of tags assigned to the document by the user.

4.7.3 The Feedbacks Core

The Feedbacks core contains feedback data regarding recommendations and their respective evaluation metrics. This information serves for recommender evaluations as, for instance, implemented in the form of A/B testing. Typically, the Feedbacks core is stable over different target domains and fields remain as listed in Table 42.

TABLE 42: FEEDBACKS CORE - FIELD DESCRIPTION

Fieldname	Description
Id	The ID of the recommendation.
recomm_profile_name	The name of the profile in the Recommender Customizer module used for generating the recommendation.
recomm_ids	An array of IDs indicating the items which were recommended.
item_ids	An array of IDs indicating the items on which the recommendation is based on.
hybrid_recomm_* [10]	These fields contain additional properties of the recommendation algorithm.
user_id	The ID of the user who received the recommendation.
custom_filters	The recommendation filter specified on the client side used for filtering the results.
recomm_algo	The algorithm which was applied for calculating the recommendation.

[10] The Feedbacks Core contains a set of parameters starting with the suffix '*hybrid_recomm_*'.

max_recomm_results	The number of recommendations requested by the client.
recomm_type	A parameter indicating whether users or items were recommended.
recomm_time	Datetime indicating when the recommendation happened.
duration	The time it took the recommendation algorithm to finish.
eval_id	The ID of the evaluation.
expected_ids	An array of items which should have been recommended. Used for calculating evaluation metrics.
interaction_count	The number of interactions resulting from the recommendation.

There was no need to adapt the Feedbacks core schema as the TRIPLE evaluation requirements can be satisfied by the default configuration. The field configuration and their data types are given in Figure 17.

```
    <fields>

      <field name="id"                          type="string" indexed="true" stored="true" required="true" />
      <field name="recomm_profile_name"         type="string" indexed="true" stored="true" />

      <field name="recomm_ids"                  type="string" indexed="true" stored="true" multiValued="true" />
      <field name="item_ids"                    type="string" indexed="true" stored="true" multiValued="true" />
      <field name="hybrid_recomm_profile_names" type="string" indexed="true" stored="true" multiValued="true" />
      <field name="hybrid_recomm_ids"           type="string" indexed="true" stored="true" multiValued="true" />
      <field name="hybrid_recomm_parent_id"     type="string" indexed="true" stored="true" />
      <field name="user_id"                     type="string" indexed="true" stored="true" />
      <field name="custom_filters"              type="string" indexed="true" stored="true" />
      <field name="recomm_algo"                 type="string" indexed="true" stored="true" />

      <field name="max_recomm_results"          type="int"    indexed="true" stored="true" />
      <field name="recomm_type"                 type="string" indexed="true" stored="true" />
      <field name="recomm_time"                 type="date"   indexed="true" stored="true" />
      <field name="duration"                    type="long"   indexed="true" stored="true" />

      <field name="eval_id"                     type="string" indexed="true" stored="true" />
      <field name="expected_ids"                type="string" indexed="true" stored="true" multiValued="true" />

      <!-- number of interactions which happend with this recommendation -->
      <field name="interaction_count"           type="long"   indexed="true" stored="true" />

      <!-- Solr for getting random items and storing the version of updates. -->
      <dynamicField name="random*"              type="random" indexed="true" stored="false"/>
      <field        name="_version_"            type="long"   indexed="true" stored="true" />

    </fields>
```

FIGURE 17: FEEDBACKS CORE SCHEMA

5 DESCRIPTION OF THE WORK DONE IN TASK 5.2

The work done so far in this task can be summarized to the following main points:

- **Understanding the role and possibilities of the Recommender System in GoTriple**
  The results of this process enter various sections of this document and the so far development of the RS. They are grounded on two main questions:
  (i) What kind of recommendations can support the user? This question was worked on in collaboration with WP3 (see also D 3.4) and is slightly discussed in Section 6.
  (ii) How can the RS be integrated in GoTriple? The outcome of this rather technical discussion mainly influenced the design of API and interaction sequences.
- **Integrating the Recommender System with the GoTriple main platform and its data backend**
  - Setting up server environments and service instances of the RS to provide for the GoTriple development and the GoTriple production platform which is described in Subsection 4.1.**Erreur ! Source du renvoi introuvable.**
  - Adapting the ScaR service API and backend to meet the requirements of GoTriple which is described in great detail in Sections 3 and 4.
  - Further developing the technical infrastructure of ScaR to provide better flexibility and a more stable environment, which is described in Subsection 5.1
- **Implementing a first set of recommendations as described in Subsection 5.2**

5.1 Infrastructure Development

As the TRIPLE project aims to innovate practices that promote the exploration of research and the delivery of content, the infrastructural level of services must also be considered to support the given ambition. Hence, the approach of deploying and hosting the recommendation services needed to be adapted to provide a reliable, scalable and maintainable system which can offer its functionality in light of the demands of the envisioned GoTriple platform.

The previous approach of deploying the RS was to individually host and run each contained module/service on its own, e.g., in dedicated servers, which made it dependent on both the underlying operating system and 3rd party software installed. Horizontally scaling the RS, with regard to the number of available instances of each module to cover varying service request claims, was complex in terms of interventions needed.

Therefore, the transition to containerized services (i.e., Docker containers[11]) bound together with the help of Docker Compose[12] was implemented, yielding the possibility to scale services more efficiently. From that, the infrastructure of the RS was lifted from a manually managed set of services to a compact software-package independent of the operational execution environment. Additionally, as services are now kept together in a smaller decoupled bundle, ease of maintenance is improved. From that, undertaken developments also considered the reliability of

[11] https://www.docker.com/resources/what-container
[12] https://docs.docker.com/compose/

services, as this is directly influenced by the effort for maintenance and the process to scale services based on the number of requests.

Steps to create the new setup included a range of code changes so that the communication between the modules is able to exploit Docker-based endpoints and to allow Docker to work with the configurations and data needed for recommendation services. Further, respective Docker images, which contain required 3rd party software and operating system dependencies for the services to be able to execute, have been created.

Hosting the RS together with other services from within the GoTriple platform now can be seamlessly implemented by adapting/extending the given Docker Compose configuration to the needs of the envisioned (cloud) infrastructure environment (e.g., Docker, Kubernetes[13], AWS[14], Google Cloud[15], etc.).

5.2 Recommendations in GoTriple

Currently, two types of algorithms are deployed for the GoTriple platform: (i) Collaborative Filtering (CF) [9] which is a personalized algorithm that analyses items' interaction data to find similar users. Subsequently, items of these similar users (i.e., nearest neighbours) are extracted and recommended. (ii) Content-based Filtering (CBF) [10] which calculates item similarities based on content features and recommends a ranked list of these similar items. The algorithms are described in a more general manner in Subsection 3.1.2.

To enable CF-based recommendations, which are based on users' interaction traces extracted from the GoTriple platform, different interactions are taken into account, i.e., tagging, accessing and opening research items and exporting information about these items. As the platform evolves, additional interaction possibilities can and will be included into the algorithm. To find research items of interest, items from within the traces of other users which share the same interests in research items are considered.

To be able to offer users with content-based documents recommendations, we calculate similarities to a user's currently viewed research item. These CBF recommendations are provided based on the similarity (i.e., using TF-IDF) of a research item's headlines, abstracts, full texts and sets of categorizing keywords attached to them. As the recommender services evolve, these attributes will be further analysed to understand the importance of their content for improving the relevance of recommended research items.

[13] https://kubernetes.io/

[14] https://aws.amazon.com/

[15] https://cloud.google.com/

6 NEXT STEPS AND EVOLUTIONS

At this point in time a first set of document recommendation algorithms (described in Section 5.2.) is available in GoTriple. This is aligned with the current state of platform development, available meta-data and GUI design. As there is a strong dependency of the recommender implementation on the data of the application context, further recommendation strategies for research peers and projects have been prepared but cannot be fully implemented and integrated yet.

However, within the upcoming period of the project we will - in line with the progressing development of the platform - expand the offered recommendation strategies to also suggest research projects and research peers with different algorithmic approaches as presented in more detail in Subsection 6.1.

For additional inspiration on state-of-the-art evolutions, we will moreover attend related international conferences as for instance RecSys[16], were we also plan to publish within the next phase of the project. From this, we expect insights and new trends in relation to building fair, transparent and bias aware recommendation systems. A selection of which will enter the design and further development of our ScaR framework. Subsection 6.2 provides a first glimpse on the impact of fairness in Recommender Systems and how it can be considered.

6.1 Further Development of Recommendations

In the course of work conducted by WP3 an end user workshop has been completed, which is described in further detail in D 3.4.

While the session only included seven participants, the qualitative nature of it allows us to draw some insights that will feed into the further development of the recommender. The following list is a summary of these insights:

- Frequency of desired recommendations depends on the context.
- Nature of desired recommendations depends on the research stage.
- User control is essential.
- High serendipity is important to users.
- There is a high interest in peer recommendations.

Taken this into account we will offer a bundle of different recommendation approaches, suggesting research documents, research peers and research projects. Recommendations will not be pushed on users but delivered on demand within the platform. Hence, a variety of recommendation services is provided at different entry points on the GoTriple platform. The user can navigate to dedicated recommendation tabs and thus controls its consumption as desired.

Similar to the available document recommendations the RS will offer first a standard implementation of a content-based and collaborative filtering recommendation algorithm for projects as well as for research peers. Using an A/B testing approach, the recommendation services will then be evaluated and tailored to the specific needs of researchers who engage with the GoTriple platform. This will inform the fine-tuning and adaptation of the algorithms. Building on

[16] RecSys 2020 (Online) – RecSys (acm.org)

the resulting, most accurate recommendation approaches, hybrid recommendation strategies will be developed that also incorporate the notion of fairness.

6.2 Fairness

Recommender systems traditionally train on data that has been collected within the specific application context in the past (i.e., historic data). This data is used to calculate similarities and correlations that form the basis for personalized recommendations. Because historic data is captured from human interaction, this data and consequently the recommendations are prone to all kinds of biases. This may lead to the emergence of unwanted patterns in relation to a user e.g., filter bubble effects, or a social group e.g., the unfair treatment of specific user groups [19].

The detection and mitigation of biases in Recommender Systems is an emerging and complex research field. In addition to better known classification problems, challenges arise with the calculation of repeated, varying and personally individual suggestions, with effects of ranking and often with the interest of multiple stakeholders [18].

In the recent past we have investigated means to measure and mitigate a number of different biases such as gender bias [20], popularity bias [21] and confirmation bias [22].

In GoTriple we will exploit existing expertise, providing hybrid extensions of traditional recommendation algorithms to mitigate

(i) gender and popularity bias through a fair consideration of all researchers regardless of gender, seniority, popularity and affiliation.

(ii) confirmation bias by optimizing algorithms not strictly for accuracy but also serendipity.

7 REFERENCES

[1] Konstan, J. A. (Ed.). (2004). Introduction to recommender systems: Algorithms and evaluation. ACM Trans. Inf. Syst. 22.1. Ed. By pp. 1-4.

[2] Linden, G., Smith, B., & York, J. (2003). Amazon.com recommendations: item-to-item collaborative filtering. IEEE Internet computing 7.1(pp. 76-80).

[3] Gomez-Uribe, C., & Hunt, N. (2016). The Netflix Recommender System. ACM Transactions on Management Information Systems (TMIS) 6.4, p. 13.

[4] Schafer, J. Ben, Joseph A. Konstan, and John Riedl (1999). Recommender systems in e-commerce. Proceedings of the 1st ACM conference on Electronic commerce. ACM, (pp. 158–166).

[5] Bobadilla, J., Ortega, F., Hernando, A., & Gutiérrez, A. (2013). Recommender systems survey. Knowledge-based systems, 46, 109-132.

[6] Lacic, E., Traub, M., Kowald, D., & Lex, E. (2015). ScaR: Towards a Real-Time Recommender Framework Following the Microservices Architecture. Workshop on Large Scale Recommender Systems (LSRS'2015) co-located with the 9th ACM Conference on Recommender Systems (RecSys'2015).

[7] Richardson, C. (2021). Pattern: Microservice Architecture. Retrieved August 30, 2021, from http://microservices.io/patterns/microservices.html.

[8] Bostandjiev, S., O'Donovan, J., & Höllerer, T. (2012, September). TasteWeights: a visual interactive hybrid recommender system. In Proceedings of the sixth ACM conference on Recommender systems (pp. 35-42).

[9] Schafer, J. B., Frankowski, D., Herlocker, J., & Sen, S. (2007). Collaborative filtering recommender systems. In The adaptive web (pp. 291-324). Springer, Berlin, Heidelberg.

[10] Lacic, E., Kowald, D., Eberhard, L., Trattner, C., Parra, D., & Marinho, L. B. (2013). Utilizing online social network and location-based data to recommend products and categories in online marketplaces. In Mining, Modeling, and Recommending'Things' in Social Media (pp. 96-115). Springer, Cham.

[11] Apache Software Foundation (2019, October). Apache Solr Reference Guide. https://solr.apache.org/guide/8_2/index.html.

[12] YAML Language Development Team (2021, September). Yaml ain't markup language (yaml™). https://yaml.org/spec/1.2.2.

[13] D. Miller, J. Whitlock, M. Gardiner, and R. Ratovsky (2014, September). OpenAPI Specification v2.0, https://spec.openapis.org/oas/v2.0.

[14] Swagger. OpenAPI Specification - fka Swagger RESTful API Documentation Specification. Version 2.0. Retrieved on September 22nd, 2021, https://swagger.io/specification/v2/

[15] Springfox. Spring Fox – Automated JSON API documentation for API's built with Spring. Retrieved on September 22nd, 2021, from http://springfox.github.io/springfox/.

[16] Salton, G., & Yang, C. S. (1973). On the specification of term values in automatic indexing. Journal of documentation, vol. 29, no. 4, Art. no. 4, 1973-04, doi: 10.1108/eb026562.

[17] Lacic, E., Kowald, D., Parra, D., Kahr, M., & Trattner, C. (2014, April). Towards a scalable social recommender engine for online marketplaces: The case of apache solr. In Proceedings of the 23rd International Conference on World Wide Web (pp. 817-822).

[18] Ekstrand, M. D., Das, A., Burke, R., & Diaz, F. (2021). Fairness and Discrimination in Information Access Systems. arXiv preprint arXiv:2105.05779.

[19] Baeza-Yates, R. (2020). Bias in Search and Recommender Systems. In Fourteenth ACM Conference on Recommender Systems (RecSys '20). Association for Computing Machinery, New York, NY, USA, 2. https://doi.org/10.1145/3383313.3418435

[20] Rekabsaz, N., Kopeinik, S., & Schedl, M. (2021). Societal Biases in Retrieved Contents: Measurement Framework and Adversarial Mitigation for BERT Rankers. arXiv preprint arXiv:2104.13640.

[21] Kowald, D., Schedl, M., & Lex, E. (2020). The unfairness of popularity bias in music recommendation: a reproducibility study. Advances in Information Retrieval, 12036, 35.

[22] Kopeinik, S., Lex, E., Kowald, D., Albert, D., & Seitlinger, P. (2019, September). A Real-Life School Study of Confirmation Bias and Polarisation in Information Behaviour. In European Conference on Technology Enhanced Learning (pp. 409-422). Springer, Cham.

8 APPENDIX I: RECOMMENDATION PROFILE CONFIGURATION

This appendix describes the most relevant recommendation profiles, followed by a description of the relevant parameter groups in alphabetical order.

8.1 The InteractionCfProfile

The InteractionCfProfile configures a collaborative filtering algorithm based on the interactions from many users. The underlying assumption of the collaborative filtering approach is that if person A has the same opinion as person B on an issue, A is more likely to have B's opinion on a different issue as well.

InteractionCfProfiles start with following header:

```
!!at.knowcenter.sc.recomm.common.profiles.InteractionCfProfile
```

At root level a InteractionCfProfile supports the parameters and parameter groups listed in

Table 43.

TABLE 43: SETTINGS AVAILABLE FOR THE INTERACTIONCFPROFILE

Name	Type	Description
coreNames	group	This parameter group holds the names of the Solr cores. At least one of the contained parameters must be given.
fetchUserHistoryProperties	group	This group contains settings for fetching the user history.
recommProps	boolean	This parameter defines whether recommendation properties should be filled. This functionality is required for flex filters.
recommType	string	This parameter specifies which type of entities can be recommended. Accepted values are “items”, “users” and “linkedItems”
similarItemsProperties	group	settings for how to retrieve similar items
similarUsersProperties	group	user retrieval settings

8.2 The ItemCbProfile

The ItemCbProfile configures a content-based filtering algorithm. Content-based filtering methods are based on a description of the item and a profile of the user's preferences.

ItemCbProfiles start with following header:

```
!!at.knowcenter.sc.recomm.common.profiles.ItemCbProfile
```

At root level an ItemCbProfile supports the parameters and parameter groups listed in

TABLE 44.

TABLE 44: SETTINGS AVAILABLE FOR THE ITEMCBPROFILE

Name	Type	Description
coreNames	group	This parameter group holds the names of the Solr cores. At least one of the contained parameters must be given.
fetchUserHistoryProperties	group	This group contains settings for fetching the user history.
mltProperties	group	This parameter group holds the parameters related to database similarity search.
recommProps	boolean	This parameter defines whether recomm properties should be filled. This functionality is required for flex filters.
recommType	string	This parameter specifies which type of entities can be recommended. Accepted values are “items”, “users” and “linkedItems”
similarItemsProperties	group	settings for how to retrieve similar items

8.3 Parameter groups

The coreNames parameter group is used in all non-hybrid recommendation profiles at root level. It specifies which backend database cores hold which type of objects. The parameters for all used cores must be set. The contents of the coreNames parameter group are described in

TABLE 45.

TABLE 45: THE CORENAMES PARAMETER GROUP.

Name	Type	Description
feedback	string	Name of the feedbacks core.
interaction	string	Name of the interactions core.
item	string	Name of the items core.
user	string	Name of the users core.

The fetchUserHistoryProperties parameter group is used in all non-hybrid recommendation profiles at root level. The contents of the fetchUserHistoryProperties parameter group are described in

TABLE 46.

TABLE 46: THE FETCHUSERHISTORYPROPERTIES PARAMETER GROUP.

Name	Type	Description
fetch	boolean	If set, to true then the user's history data will be included (i.e., user's preferences)
interactionCoreProperties	group	Contains properties for fetching the user history based on the interaction core.
itemCount	integer	The number of items from the user's history which should be considered.

The flexFilters parameter group specifies a set of filters to define a subset of items. Parameters contained in the flexFilters parameter group are listed in

Table 47.

TABLE 47: THE FLEXFILTERS PARAMETER GROUP.

Name	Type	Description

fieldFilters	list of fieldFilters	Filters items based on a list of field name-value pairs.
		
		Example for the inclusion of items with the type "DOCUMENT":
		
		`fieldFilters`
		`  - fieldName: type`
		`    fieldValue: DOCUMENT`

The interactionCoreProperties parameter group contains properties for fetching the user history based on the interaction core. The possible entries are described in

TABLE 48.

TABLE 48: THE INTERACTIONCOREPROPERTIES PARAMETER GROUP.

Name	Type	Description
fetchType	string	Defines how documents are fetched from the interactions core. Possible values are "SORTED_TIME_BASED" and "FACETED_ITEM_COUNT_BASED".
interactionCount	integer	The number of user interactions to query for item extraction.
interactionTypes	list of strings	Interaction types to include.
itemFieldName	string	Field in the interaction core to include which identifies an

		item.
timeFieldName	string	Field to be observed if the time component of user interactions is important.
userFieldNames	list of strings	Fields in the interaction core to include which identify users.

The mltProperties parameter group holds parameters related to database similarity search. Available entries are described in

Table 49.

TABLE 49: THE MLTPROPERTIES PARAMETER GROUP.

Name	Type	Description
mindf	integer	Specifies the Minimum Document Frequency, the frequency at which words will be ignored which do not occur in at least this many documents.
mintf	integer	Specifies the Minimum Term Frequency, the frequency below which terms will be ignored in the source document.
minwl	integer	Sets the minimum word length below which words will be ignored.
maxqt	integer	Sets the maximum number of query terms that will be included in any generated query.
maxResultsPerItem	integer	Number of documents to include in the result of the similarity query.

queries	list	List of boosting options: The importance of a field can be "boosted". A boosting option contains a boosting factor ("boost") and a field name ("field").
		
		Example for boosting the fields "title" and "full_text" with different boosting factors:
		
		`queries:`
		`  - boost: 1.0`
		`    field: title`
		`  - boost: 1.5`
		`    field: fullt_text`

The similarItemsProperties parameter group can be used in non-hybrid recommendation profiles at root level when item similarities are used in the recommendation algorithm. Available entries for this parameter group are described in

TABLE 50.

TABLE 50: THE SIMILARITEMSPROPERTIES PARAMETER GROUP.

Name	Type	Description
filterUserItems	boolean	Whether to include user's preference data (e.g. users interactions with items of interest).
flexFilters	list of flexFilters	Contains a list of filters, which define a subset of items to include. See group "flexFilters" for details.

interactionFields	group	User interaction fields to use. This group contains only the parameters "fields", which holds a list of field names.
		
		Example with four fields:
		
		`interactionFields:`
		`  fields:`
		`  - name: users_tag_document`
		`  - name: users_access_document`
		`  - name: users_export_information`
		`  - name: users_open_descriptio`

usersToIgnore	list of strings	A list of user IDs to ignore. Typically used to filter out dummy users.

The *similarUsersProperties* parameter group defines the condition for the retrieval of similar users. These settings are relevant when an algorithm uses user similarities. Possible values are described in Table 51.

TABLE 51: THE SIMILARUSERSPROPERTIES PARAMETER GROUP.

Name	Type	Description
checkWhetherUsersExist	boolean	Check whether the users exist for which the interactions were recorded.
minItemOverlap	integer	Minimal number of overlapping items to conclude that 2 users are similar.
similarUserCount	integer	Maximal number of users included in the result of the similarity calculation.
topKSimilarUserCount	integer	Maximum number of uses to include in the result of top K user similarity calculation.

8.3.1 Personalized research projects

The recommender profile for recommending personalized research projects is configured in the file triple_project_cf.yaml whose contents are shown in Figure Figure 18: triple_project_cf.yaml. The profile resembles the profile for recommending personalized research documents but uses interactions that can only be done on research projects. The profile uses the InteractionCfProfile which is a profile for the Collaborative Filtering algorithm. It is configured to recommend entities from the items core which is named “items”. Up to 10 items from the user’s history are considered. These items are extracted from the “document_id” field of the user’s last 40 interactions. Users are identified either by their user ID or their session ID. Project specific interaction types are taken into account. These are “OPEN_PROJECT_DESCRIPTION”, “ACCESS_PROJECT”, and “EXPORT_PROJECT_INFORMATION”. Up to 40 similar users are considered for generating recommendations. Users must have at least one common item to be considered for similarity calculations. Similar items are considered when they are of type “PROJECT”. Item similarity calculations are based on the project specific interaction fields “users_access_project”, “users_export_project_information”, and “users_open_project_description”.

```
!!at.knowcenter.sc.recomm.common.profiles.InteractionCfProfile
recommType: items
coreNames:
  item: items
recommProps: true
fetchUserHistoryProperties:
  fetch: true
  itemCount: 10
  interactionCoreProperties:
    interactionCount: 40
    timeFieldName: timestamp
    userFieldNames:
      - user_id
      - session_id
    itemFieldName: project_id
    interactionTypes:
      - OPEN_PROJECT_DESCRIPTION
      - ACCESS_PROJECT
      - EXPORT_PROJECT_INFORMATION
similarUsersProperties:
  similarUserCount: 40
  topKSimilarUserCount: 40
  minItemOverlap: 1
similarItemsProperties:
  filterUserItems: true
  flexFilters:
    fieldFilters:
    - fieldName: type
      fieldValue: PROJECT
  interactionFields:
    fields:
    - name: users_access_project
    - name: users_export_project_information
    - name: users_open_project_description
  usersToIgnore: []
```

FIGURE 18: TRIPLE_PROJECT_CF.YAML

8.3.2 Similar research projects

The recommender profile for recommending similar research projects is configured in the file triple_project_item_cb.yaml whose contents are shown in Figure 19. It uses the ItemCbProfile which is a profile for the Content Based Filtering algorithm. It is configured to recommend entities from the items core which is named “items”. User history is not considered for recommendation. Similar items are identified based on the contents of following fields: “full_text”, “title”, “abstracts”, “headlines”, “keywords”, “topics”, “knows_about”, “disciplines”, and “similar_subjects”. All fields are considered with the same importance. Words shorter than three characters are ignored and there are no limitations on term frequency and document frequency. The length of the generated similarity query is limited to 25 query terms and the query result is limited to 10 items. Similar items are considered for recommendation when they are of type “PROJECT”.

```
!!at.knowcenter.sc.recomm.common.profiles.ItemCbProfile
recommType: items
recommProps: true
coreNames:
  item: items
fetchUserHistoryProperties:
  fetch: false
mltProperties:
  maxqt: 25
  mindf: 1
  mintf: 1
  minwl: 3
  queries:
  - boost: 1.0
    field: full_text
  - boost: 1.0
    field: title
  - boost: 1.0
    field: abstracts
  - boost: 1.0
    field: headlines
  - boost: 1.0
    field: keywords
  - boost: 1.0
    field: topics
  - boost: 1.0
    field: knows_about
  - boost: 1.0
    field: disciplines
  - boost: 1.0
    field: similar_subjects
  maxResultsPerItem: 10
similarItemsProperties:
  filterUserItems: false
  flexFilters:
    fieldFilters:
    - fieldName: type
      fieldValue: PROJECT
  usersToIgnore: []
```

FIGURE 19: TRIPLE_PROJECT_ITEM_CB.YAML

Research peers

The recommender profile for recommending research peers is another example for Collaborative Filtering. The profile is configured in the file triple_user_cf.yaml and uses the ItemCbProfile. File contents but is not finished yet.

9 APPENDIX II: INSTALLATION AND CONFIGURATION

9.1 Installation Guide

This section gives an introduction to the deployment and configuration procedure. The system is delivered and deployed with a working configuration.

PREREQUISITES

ScaR requires Docker and Docker-Compose to be installed on the host system.

FOLDER STRUCTURE

Figure 20 gives an overview of the folder structure on the server. The configuration files are located in the `conf` folder, the recommender profiles are located in `conf/profiles`, and the Solr database configuration goes in the `solr` folder. The Docker configuration is contained in the Docker Compose file.

```
.                           project root
├── conf                    configuration folder
│   ├── dml.yaml            DML configuration file
│   ├── engine.yaml         Recommender Engine configuration
│   ├── profiles            Folder for recommender profiles
│   ├── repo.yaml           Recommender Customizer config
│   └── security.json       Solr security configuration
├── docker-compose.yml      Docker compose file
└── solr                    Solr database
    ├── feedbacks           Solr feedbacks core
    ├── interactions        Solr interactions core
    └── items               Solr items core
```

FIGURE 20: FOLDER STRUCTURE.

DEPLOYMENT PROCEDURE

1. Stop ScaR

   For proper re-deployment or modification all affected modules must be stopped. This is especially important if the changes affect the backend database in any way.

   To stop all services at once, change into the project root folder and execute:

   ```
   docker-compose down
   ```

2. Get and install the Docker images

   This step varies depending on the project. It depends on the delivery modalities agreed between the project partners.

3. Database configuration

The system is shipped with a working database configuration. Copy the provided database configuration in the `solr` folder before the first run. It is important that the Solr database service has complete read and write permissions for the database folder. This condition must also hold for the user named “solr” inside the Docker container. The most effective method to achieve this is to make the solr directory with its content completely "open" in terms of read/write permissions. To achieve this, change to the project root folder and execute:

```
sudo chmod -R 777 solr
```

4. Copy the (updated) configuration to the `conf` folder.
5. Copy the (updated) recommender profiles to the `conf/profiles` folder.

   Note: The **`conf/profiles`** directory *must not* contain other files than the recommendation profiles.

6. Copy the (updated) `docker-compose.yml` file to the project root folder.
7. To start ScaR, enter the project root folder and execute the final command:

```
docker-compose up
```

RECOMMENDER CONFIGURATION FILES

- dml.yaml: The configuration file for the Data Modification Layer
- engine.yaml: The configuration file for the Recommender Engine
- repo.yaml: The configuration file for the Recommender Customizer.

DATABASE ACCESS CONFIGURATION

- security.json: Solr’s configuration file for authentication and authorization. Please consult the official Solr documentation [11].

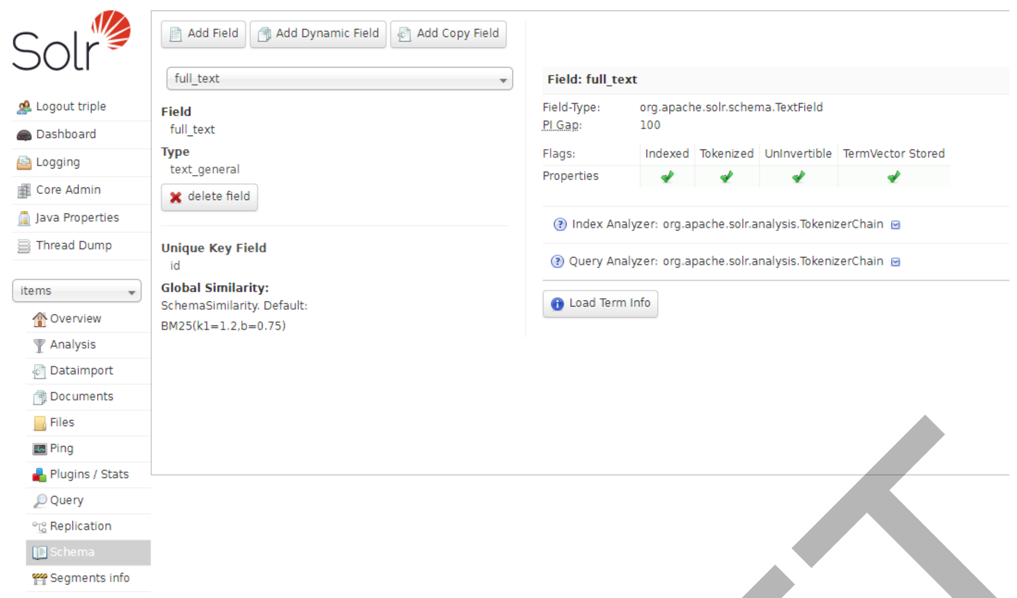
RECOMMENDATION PROFILES

The system is delivered with preconfigured recommendation profiles. A detailed description can be found in Appendix I: Recommendation Profile Configuration.

9.2 Database Configuration

This section gives a short overview on the methods used to configure the Solr database. For detailed in depth documentation of Solr configuration please consult the official Solr Reference Guide [11].

Solr provides an administrative web interface, where the database can be configured with a comfortable graphical user interface. All operations necessary for maintenance and configuration can be performed here. Figure 21 shows a screenshot of the administrative web interface.



Field
full_text
Type
text_general

Unique Key Field
id
Global Similarity:
SchemaSimilarity. Default: BM25(k1=1.2,b=0.75)

Field: full_text

Field-Type:	org.apache.solr.schema.TextField			
PI Gap:	100			
Flags:	Indexed	Tokenized	UnInvertible	TermVector Stored
Properties	✓	✓	✓	✓

Index Analyzer: org.apache.solr.analysis.TokenizerChain

Query Analyzer: org.apache.solr.analysis.TokenizerChain

FIGURE 21: SOLR ADMIN GUI

The Solr Admin GUI comes in handy for quick configuration updates, but initial database configuration becomes tedious via the admin GUI. Therefore, the core schemas are configured in the associated schema.xml file. These schema configuration files are matched to the target domain during system integration and are delivered together with the software.