Complex Systems Informatics and Modeling Quarterly
CSIMQ, Issue 9, December 2016/January 2017, Pages 67–83
Published online by RTU Press, https://csimq-journals.rtu.lv
https://doi.org/10.7250/csimq.2016-9.04
ISSN: 2255-9922 online

Metadata Extraction and Management in Data Lakes With GEMMS

Christoph Quix[1,2⋆], Rihan Hai[2] and Ivan Vatov[2]

[1]Fraunhofer-Institute for Applied Information Technology FIT,
Schloss Birlinghoven 53754 Sankt Augustin, Germany
[2]Databases and Information Systems, RWTH Aachen University,
Templergraben 55, 52062 Aachen, Germany

christoph.quix@fit.fraunhofer.de (orcid.org/0000-0002-1698-4345),
hai@dbis.rwth-aachen.de, ivan.vatov@rwth-aachen.de

Abstract. In addition to volume and velocity, Big data is also characterized by its variety. Variety in structure and semantics requires new integration approaches which can resolve the integration challenges also for large volumes of data. Data lakes should reduce the upfront integration costs and provide a more flexible way for data integration and analysis, as source data is loaded in its original structure to the data lake repository. Some syntactic transformation might be applied to enable access to the data in one common repository; however, a deep semantic integration is done only after the initial loading of the data into the data lake. Thereby, data is easily made available and can be restructured, aggregated, and transformed as required by later applications. Metadata management is a crucial component in a data lake, as the source data needs to be described by metadata to capture its semantics. We developed a Generic and Extensible Metadata Management System for data lakes (called GEMMS) that aims at the automatic extraction of metadata from a wide variety of data sources. Furthermore, the metadata is managed in an extensible metamodel that distinguishes structural and semantical metadata. The use case applied for evaluation is from the life science domain where the data is often stored only in files which hinders data access and efficient querying. The GEMMS framework has been proven to be useful in this domain. Especially, the extensibility and flexibility of the framework are important, as data and metadata structures in scientific experiments cannot be defined *a priori*.

Keywords: Metadata management, data integration, scientific data, metadata extraction, data lakes.

1 Introduction

An increasing amount of data is generated today by various users, sensors, or systems. Although the data is available and accessible, its processing and analysis is still often a manual task which requires a lot of human guidance and control. Data has to be extracted from the data sources, it has

⋆ Corresponding author

Reference: C. Quix, R. Hai and I. Vatov, “Metadata Extraction and Management in Data Lakes With GEMMS,” Complex Systems Informatics and Modeling Quarterly, CSIMQ, no. 9, pp. 67–83, 2016. [Online]. Available: https://doi.org/10.7250/csimq.2016-9.04

be to cleaned, transformed, and mapped to a target system, and finally it has to be loaded into a data management system where it can be integrated with other data. This is known as ETL process (Extract-Transform-Load [1]) and works well for rather static data integration workflows (e.g., in data warehouse systems). Recently, the integration process is becoming more an ELT process, i.e., after extraction, the data is loaded into a central repository, and the cleaning and transformation steps are done within the repository [2]. Data lakes follow the same idea: data is extracted from the sources and is stored in its original structure in a repository which is often based on Hadoop or NoSQL database systems [3], [4]. These approaches focus on more dynamic environments, where sources and schemas are frequently changing and cannot be completely known in advance. Their advantage is that they reduce the integration efforts which have to be spent *before* the repository can be used.

Scientific applications are an example in which flexible data management and integration solutions are required [5], [6]. Especially, in the life science domain, data from experiments is collected and processed in various files (e.g., CSV, Excel, or proprietary file formats of software tools or hardware devices) using a broad range of tools (image analysis, statistics, and data mining with tools such as MATLAB or R). There are no predefined schemas, standards are rarely used, and the whole workflow is documented only (if at all) in a lab notebook in an unstructured form. To avoid repeated experiments for the same substances, or to learn from other similar experiments, an integration of this data would be very beneficial for the scientists as they could explore, query, and analyze the data of previous experiments in order to improve the setup of their next experiments [7]. However, such an analysis is not possible if there is no integrated data repository. Building an integrated data repository for a wide range of scientific data is a challenge because of the lack of well-defined schemas and frequently changing requirements. A metadata repository would be already very helpful which should contain descriptions of the data sources (or data files) available.

Date	09/2015					
Autor	John Doe					
Label: Label1						
Mode				Measurement from above		
Emission wavelength start				380	nm	
Emission wavelength end				600	nm	
Emissions wavelength step				2	nm	
Scan count				111		
Spectrum (Em)				280...850: 20 nm		
Spectrum (ex) (Sector 1)				230...315: 5 nm		
Spectrum (ex) (Sector 2)				316...850: 10 nm		
	Temperature: 25.5 °C					
WL	380	382	384	386	388	390
E1	966	224	162	171	206	273
E2	477	240	135	168	148	150
E3	627	235	171	174	232	263
E4	280	160	147	214	252	375
E5	657	245	164	167	157	179
E6	159	97	95	101	150	171

Figure 1. Spreadsheet Data Structure Example

Figure 1 shows a typical example for a data file in the life science domain. It is a spreadsheet file which has been generated by the control software of some hardware device. The spreadsheet is self-describing and contains metadata about the experiment (author, date, what was measured, parameter values, etc.), as well as the measured data with its schema (e.g., headers in columns and rows). Note that this is an adapted example and in fact both the raw data matrix and the metadata properties are a lot larger. Other data files may have a different set of metadata properties,

various data structures (e.g., table- or tree-like structures instead of two-dimensional matrices), multiple data units (e.g., several sheets within one Excel file), or completely different syntactical formats (e.g., CSV, XML, JSON). Metadata might be encoded inside the file, in the filename, or the directories of the file system. This heterogeneity of managing metadata and data makes it very hard for scientists to search for data efficiently. Usually, keyword queries across the file system are the major method for searching for information. Data management is done in scripts by reading and writing text-based data files.

To support the scientist in her data management activities efficiently and effectively, a system should provide more sophisticated metadata management functionalities which include especially an interface that allows queries over (at least semi-)structured data and metadata. The metadata should include structural information that describes the schema of the data, but also information about the semantics of the metadata and data elements. Semantical information is important as often acronyms or synonyms are used as the name of a schema element. By using semantic annotations, such ambiguities can be avoided and the semantics of elements can be clearly defined. Especially in the life sciences, ontologies are frequently used to standardize terminologies.

In this article, we describe the design, implementation, and evaluation of a **G**eneric and **E**xtensible **M**etadata **M**anagement **S**ystem (GEMMS) which (i) extracts data and metadata from heterogeneous sources, (ii) stores the metadata in an extensible metamodel, (iii) enables the annotation of the metadata with semantic information, and (iv) provides basic querying support. The system should be also flexible and extensible, as new types of sources should be easily integrated, which we prove in the evaluation. GEMMS is a major component in a data lake system for scientific data in the life science domain, which we are currently developing in the HUMIT project[1]. The project aims at an interactive and incremental integration approach in which a user can browse and query a data lake and define mappings. The application domain of the project is life science research, but the approach could be also applied to other domains. An overview of our approach is shown Figure 2.

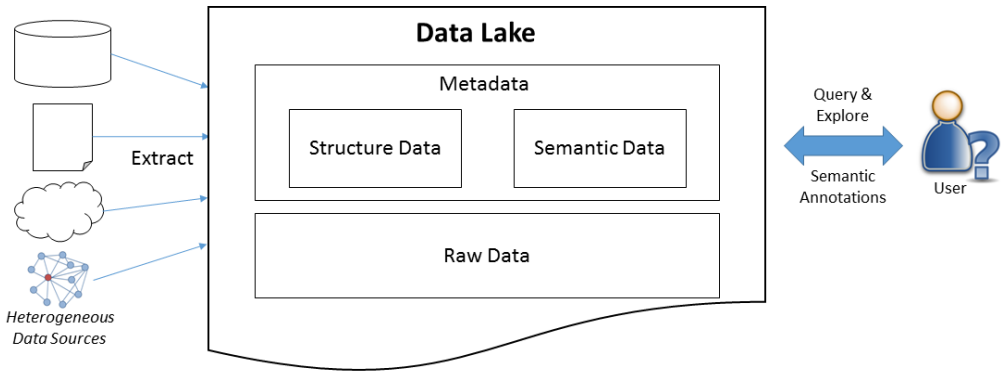

Figure 2. System overview

The article is structured as follows. Section 2 discusses the existing work on incremental integration systems. Section 3 introduces our metadata model, while Section 4 describes our system and an algorithm for deriving a tree model from a semi-structured data source. In Section 5, we evaluate our system with respect to extensibility and performance. The Section 6 concludes the article and gives an outlook.

[1] See http://www.humit.de or [8].

2 Related Work

2.1 Data Lakes

Being a relatively young concept, the data lake has not yet been extensively researched. Data lakes should store large parts of organization's data, regardless of format and without requiring an extensive schema design [4]. Data transformation or aggregation should be only applied after the data has been loaded to a repository, as early transformations or aggregations might limit the applicability of the data. Metadata is frequently pointed out as the key to describe and navigate through the massive content of a data lake [9], but details about the functional requirements and design of a metadata management component in data lakes are missing. Therefore, we address them in this article.

First data lake systems are described in [4], [10] and [11]. Google's Goods approach [10] is similar to our approach as it also focuses on the collection of metadata for the datasets which are available inside the organization. IBM's approach [4] also deals with data transformation (wrangling and curation). However, both systems do not aim at extracting different metadata information automatically from various data sources, including semi-structured files. Boci et al. [11] present a technical architecture for a data lake system based on the Hadoop eco-system, which is applied in the area of flight tracking.

2.2 Dataspaces

A similar incremental integration approach as data lakes was introduced earlier with *dataspaces* [12] which feature human defined mappings between different sources, defined in a pay-as-you-go fashion. Mappings are defined only if they are required for a specific integration task; as in data lakes, the source data stays in its original format [13]. Dataspaces and data lakes share the same goal of using metadata to search over data sources, regardless of matter of their format. However, metadata management was not considered a core functionality of the envisioned dataspace systems, which is in contrast to our approach and the data lake idea.

An implementation of the dataspace concept is *Personal Information Management (PIP)* system, also referred as *iMeMex* [14]. It manages personal information and provides functions such as searching the user dataspace for a high variety of information. As in data lakes, the raw source data is also kept in its original format.

GEMMS implements the data lake vision by automatically extracting metadata from data sources and storing it in an extensible metamodel.

2.3 Model Management

Metadata management is strongly related to model management [15], which aims at providing formal operators for models and mappings. The model representation which we apply in this approach is less detailed than in model management; e.g., in our previous work [16], we developed a generic metamodel for model management operations. This level of detail is not required here as we focus on data processing and not on operations on data models. For the representation of mappings, GEMMS is certainly less formal than the most of model management approaches, as mappings are not yet explicitly modeled and only simple semantic annotations are possible. More complex mappings are planned for the future, which will be based on our experience with nested mappings for generic model management [17]. Other model management aspects, such as schema matching [18] or schema summarization [19], are also relevant for data lakes, as extracted models need to be matched and consolidated.

2.4 Semantic Annotations

Life science is a research area in which ontologies and other concepts of the semantic web are used already for a long time [5], [20]. Many of the datasets which are available online in this domain are ontologies or linked open datasets expressed in RDF. The Gene Ontology was one of the first very rich ontologies in this domain [20], and it is still being maintained and extended. Many ontologies can be found using domain-specific search engines, such as BioPortal[2], which also provides mappings between different ontologies. This also reveals a common problem in the use of the semantic web technologies: although the technologies could be used to create a common, *shared* knowledge base, many data sets do not make use of the feature to link to or reuse other existing data sets/ontologies. This leads to the situation, that the same concept is described in different ontologies with different identifiers. This is also supported by the recommended practice: data sets should be converted to RDF to make data accessible for other users[3].

This problem and the fact that a huge amount of data is originally not available in RDF (e.g., relational databases, spreadsheets, CSVs, XML, or JSON files) has led to the idea that data should be annotated with semantics rather than translated into another format. Schema.org [21] is an initiative to create a common, well-curated, light-weight data model that can be used to annotate existing structured and unstructured data. These annotations can be also integrated into HTML pages and email messages and are being exploited by various applications from Google, Microsoft, and others. We follow here a similar idea, we use a light-weight approach to annotate existing data to simplify the exploration and usage of the data.

3 Metadata Model

Our motivating example in Figure 1 illustrates that data sources come with different types of metadata. There is descriptive metadata in the header of the file which gives more information about the contents of the source. It is usually just a list of key-value pairs which does not follow a strict model. Values are either simple literals or could have also a complex structure (e.g., value ranges such as '280–850' in Figure 1). We will model this type of metadata as *metadata properties*.

More important for the extraction and integration of data is the structural information of the source, i.e., what is the structure of the raw data contained in the source. In the example from Figure 1, the raw data is contained in a matrix, but other data structures such as trees, graphs, or simple tables are also possible. Therefore, we must be able to describe the various data structures which might appear in a data source. In the example, we should model the information that the matrix has two dimensions, and that $E1, E2, \ldots$ and $380, 382, \ldots$ are values in these dimensions. This description can then later be used for mapping the source data to another data structure or to formulate a query. We model this data as *structure metadata* in our approach.

Metadata properties and structure metadata are important elements to describe a data source, but are only of limited use if we do not understand the names which are used for properties or metadata elements. Furthermore, labels might be understandable for humans, but a system has to have a more explicit representation of the semantic information. Therefore, our metamodel allows the annotation of metadata with *semantic data*, which link the 'plain' metadata objects to elements from a semantic model (e.g., an ontology).

In the following, we describe these metadata types in more detail.

[2] http://bioportal.bioontology.org/

[3] https://www.w3.org/2001/sw/hcls/notes/hcls-rdf-guide/

3.1 Metadata Types

Structure data[4] should provide information for navigating through the raw data and representing individual models in data sources. To guarantee the genericness and extensibility in our approach, the source data encountered in the source files retain their own model instead of being matched to some predefined model. Moreover, representing the schema of each data source in its specific schema language would eventually make querying the data lake very hard or even hinder it completely, as for each source a different type of modeling and query language has to be taken into account. Therefore, instead of constructing the structure using various schema languages (e.g., XSD or SQL DDL), the approach taken is to use a unified representation similar, but simpler than the generic modeling framework *GeRoMe* which we developed earlier [16]. In *GeRoMe*, we had a very detailed representation of all features and constraints; even simple types restricted by regular expressions as in XML Schema could be represented in *GeRoMe*. We decided not to support such very detailed constraints in this framework, as it is not necessary for the exploration of the available data in the data lake. In this context, it is sufficient to have information about the structure of a data source and its basic data types.

We formalized two subtypes of structure data in GEMMS – tree structures and matrix structures. Data from the source, which is inherently tree-like (e.g., an XML document) is converted to an internal abstract tree with the same structure. As we focus on the extraction of metadata, we retain only element names and cardinalities. Details about the algorithms for inferring the structure data from the source data are given in Section 4.3. Again, we use here a simpler approach than other approaches for schema-extraction from semi-structured data (e.g., [22] or [23]), as we just want to produce a comprehensive overview of the extracted data, and not a rigid schema to validate data and ensure consistency of a database. Structure data can be also specified explicitly for a specific file type (e.g., if all files with this type have the same structure) or it could be also completed by user input.

Metadata properties are a set of customizable properties for each data source. There are metadata properties provided by the file system, such as filename, size, location or date of last modification. On the other hand, some metadata fields are part of the file contents which are highly specific for each file type, e.g., the date or the author of an experiment. This type of metadata fields are present along with the raw data, but in essence are metadata as key-value pairs which *describe* the raw data. Therefore, they are of interest for our metadata management system. Their number is arbitrary for each file, so metadata properties are not part of the structure data. As the metadata properties and their names can be very heterogeneous in different data sources, we provide the possibility to annotate the metadata with semantic data.

Semantic Data handles the *meaning* of the entities, described throughout the data sources. The general idea is to allow annotations to be attached to different model elements. For instance, a certain data source can be attached to a list of annotations, the same way as a dimensional value of a matrix can be attached to a list of annotations, or even a node from the tree structure data. We are not making any restrictions on the type of annotations which are allowed for a model element; the annotation should make a reference to a clearly identified element. The simplest form of annotation just uses an URI (Universal Resource Identifier) that refers to some ontology term. However, also other types of identifiers could be used, as long as they are somehow unique and it is clear which element is referenced.

The idea of these semantic annotations is that the user will do her querying using such elements with a well-defined semantics instead of using the labels of metadata properties or tree elements directly. Thus, the ontology to be used in the semantic annotations should be specific for the domain

[4] In a strict manner, structure data should be replaced by the term *structure metadata*. However, in order not to abuse the term metadata, structure data is used in the rest of the article.

of the data lake system. In the context of the HUMIT project, for instance, we plan to use the BioAssay Ontology [24] as it represents quite well the elements which are used in our data sources.

Contrary to the structure data, semantic data cannot be inferred, so it is expected that annotations are explicitly specified for a file type or done manually by a domain expert. As one of our future works, the possibility of using classification algorithms over the set of already annotated data attributes could be investigated. New data sources could then be automatically annotated based on the annotations already available.

3.2 Conceptual View of the Metadata Model

Figure 3 depicts a high-level view over our metadata model using a simplified notation of UML class diagrams. All relationships between the model elements are of the "weak" has-a relationship (or aggregation). More specifically, some of the relationships are actually compositions, but it was decided against the implied constraints of this kind of relationship. Four of the relationships are modeled as annotation relationships, i.e., the semantic annotations are represented as parts of the other model elements.

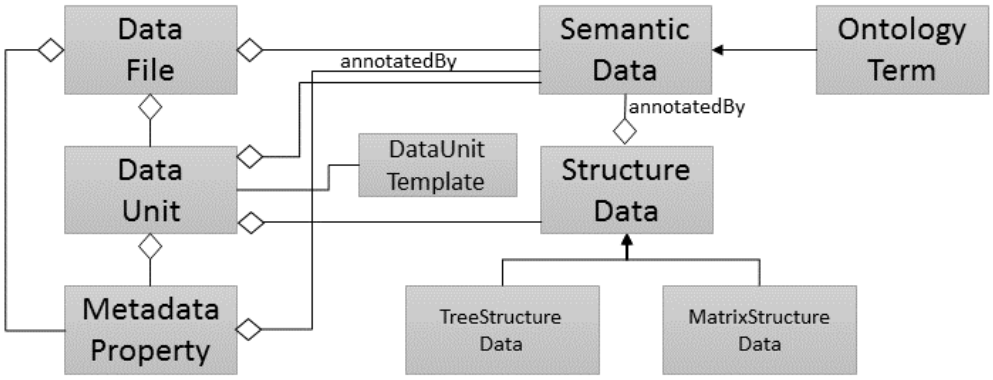

Figure 3. Conceptual View of the Model

The model elements *Data File* and *Data Unit* have not been discussed so far. In this article, as we mainly focus on accommodating files as data sources, we use the *Data File* element as one model component to present the metadata of a file. We are currently working on the extension of our approach to handle also general data sources (e.g., database systems or web services). Thus, in the future, this concept will be generalized to an element *Data Source*.

As the main part of our model, a *Data Unit* represents an independent piece of data, which might carry its own metadata and raw data. A data unit contains most of the relevant metadata information and is flexible enough for other types of data sources as well. The data unit is an abstract entity, containing the structure of the data it contains, plus additional metadata properties. As all other elements, the data unit can be also annotated. Furthermore, for each file type, the scope of a data unit and which metadata properties are part of a data unit can be defined. This information is provided by a *Data Unit Template*. For different file formats, the data unit has different semantics. For instance, in Excel files, data units represent worksheets; in an XML document, different data units could represent different sub-trees of the whole XML document tree. In a relational database system, a data unit could be a database or table space. The main two advantages of applying data units are that they give the user flexibility during the data ingestion process, and also provide a level of abstraction above data files. Data files, as well as other data sources in general, can be seen as containers for data units, since the latter are the ones most relevant for the metadata.

With regard to the relationship between *structure data* and *data unit*, the structure data is attached to a data unit, since it carries the raw data, whose schema should be remembered. The cardinality of the relationship is one to one, indicating that each data unit has at most one structure data element.

The top-level structure data element can hold more tree or matrix structure elements, depending on the actual type of the data it models.

Metadata properties are part of both, a data unit and a data file. In the first case, metadata properties model custom in-file metadata, which the user has to specify beforehand, while in the second case they are automatically extracted from each file and contain file system attributes plus the media type. Although file system attributes and the media type are usually specific for the data sources, and do not provide much (or any) insight into the data of the files, they are nevertheless mandatory for revision and reproducibility purposes.

As summary, the data model described in this section performs two main tasks: (1) capture the general metadata properties in the form of key-value pairs, as well as structure data to aid in future querying, and (2) attach annotations (usually represented as URIs to ontology elements) to metadata elements.

4 System Design

We divide the functionalities of GEMMS into three parts: metadata extraction, transformation of the metadata to the metadata model, and metadata storage in a data store. Design and implementation of the system aim at extensibility and flexibility. Future changes, such as new data sources with yet unknown data structures and interfaces, should be realized with a minimum overhead. The system design follows SOLID principles for object-oriented software construction [25]. In this section we give an overview of the system architecture, with following subsections sequentially elaborating on the details.

4.1 System Architecture

The high-level design of the system is depicted in Figure 4. The implementation of each component contains multiple classes. Even each relationship has a 'uses' role, the most highly coupled component is the metadata manager, which orchestrates the whole process. Another self-contained module is the *extractor*, which uses a module for media type file detection and a component parsing files. The components are described in more detailed in the following.

The *Metadata Manager* invokes the functions of the other modules and controls the whole ingestion process. It is usually invoked at the arrival of new files, either explicitly by a user using the command-line interface or by a regularly scheduled job. The metadata manager reads its parameters and task from a configuration file and then starts processing the input files.

With the assistance of the *Media Type Detector* and the *Parser Component*, the *Extractor Component* extracts the metadata from files. Given an input file, the *Media Type Detector* detects its format, returns the information to the *Extractor Component*, which instantiates a corresponding

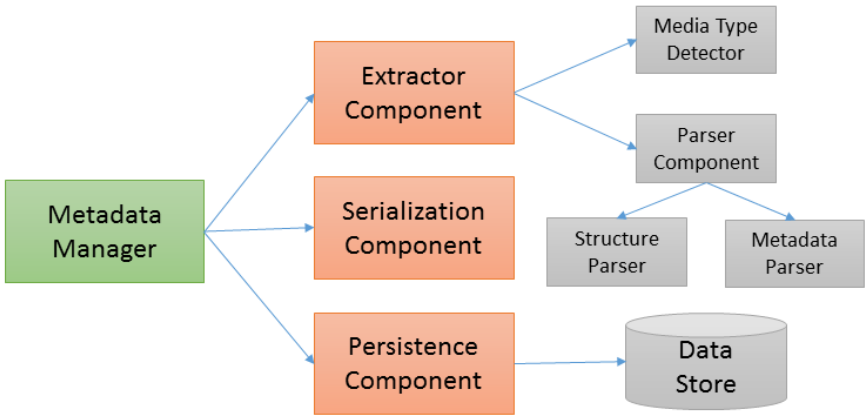

Figure 4. Overview of the system architecture

Parser Component. The media type detector is based to a large degree on Apache Tika[5], a framework for the detection of file types and extraction of metadata and data for a large number of file types. Media type detection will first investigate the file extension, but as this might be too generic (e.g., for XML files), it is possible to refine the detection strategy by specifying byte patterns (which should occur at the beginning of a file) or by providing custom detector classes. Tika provides detection mechanisms for many standard file types, but it can be also extended with new file types. We made use of this extension mechanism to implement detector classes for some life-science-specific file types (see Section 5).

When the type of input file is known, the *Parser Component* can read the inner structure of the file and extract all the needed metadata. Every parser understands the file type and data structure of the file which it is built for, and takes care about specific metadata – either structure data or custom metadata properties. Note that the high expressiveness of some formats, such as XML, implies the existence of multiple parsers for the same data file type, since the *medium* is clear (e.g., XML DOM tree), but the *structure* could be entirely different. The main distinction between extractor and parser components is that the extractor module manages different types of metadata, e.g., structure data or metadata properties, while the parser performs the actual file reading and is specialized in a single type and file structure. The parser uses third-party frameworks working on a lower level than Tika (e.g., Apache POI).

The parsers also make use of several algorithms, for instance, to detect a matrix structure inside a spreadsheet (as in Figure 1) or to create an abstract description of a tree structure (i.e., a structure similar to a DTD or an XML Schema). Due to space constraints, we can only present one of these algorithms in the article; the tree structure inference algorithm is described in more detail below in Section 4.3.

One important connection point between the data model and the system components is the *Data Unit Template*. It is used to define what information should be extracted and the module using it most actively is the parser. Intuitively, a data unit template gives details about the metadata needed from each data file type (cf. Section 3). The parser will use the template to instantiate a corresponding data unit, and then fill this data unit with the metadata extracted from the file. Data unit templates can be more specific than a file type. For instance, there is one file type for Excel files, but there can be several data unit templates, each one specifying a different set of metadata properties to be extracted from the spreadsheet (the metadata properties in the header of a sheet can be different for each file). For XML documents, the data unit templates contains XPath expressions which specify the location of data units and metadata in the XML file.

The *Persistence Component* accesses the data storage available for GEMMS. As a facade, it shields the intricacies of the concrete storage solution and provides a common and unified interface. The *Serialization Component* performs the transformation between models and the storage format. As the serialized objects have to be handled by the storage engine, it is closely connected to the persistence component. In our current implementation, we use JSON as serialization format and MongoDB as storage engine.

4.2 System Behavior

In this subsection, we discuss about behavioral rules followed in the communication between the system components, and the sequence of the messages passed among them, as shown on Figure 5.

The process is initiated by the user who inputs a configuration file through the command line interface. This file contains the data unit templates for the file types of interest, and a list of directories which has to be traversed for data files. The metadata manager then uses the serialization component to parse the file to obtain a list of files and a list of data unit templates. The extractor component is inquired about a specific extractor for each of the files from the list and it in turn

[5] http://tika.apache.org

asks the media type detector to do the job for it, providing it with the current data file. Please note that the mapping between media types and the extractors responsible is fully customizable in a configuration file. So this configuration file is used by the media type detector which instantiates an applicable extractor, which is eventually returned to its client – the metadata manager.

Once a specific extractor is available, it extracts the data from all the files it is responsible for. The most notable parameters of this invocation are the data file and all the data unit templates given for the media type of the file. Each extractor extracts structure data and metadata properties using parsers suitable for the aim. The knowledge which parser is responsible for which metadata and file type is contained in the extractors. After the metadata has been collected by the extractor component, it is combined in data units and returned to the metadata manager. Recall that a single file of data could produce multiple data units, containing its metadata. Once a data file instance with its corresponding list of data units is available to the metadata manager, it serializes it via the serialization component and then passes it to the persistence component.

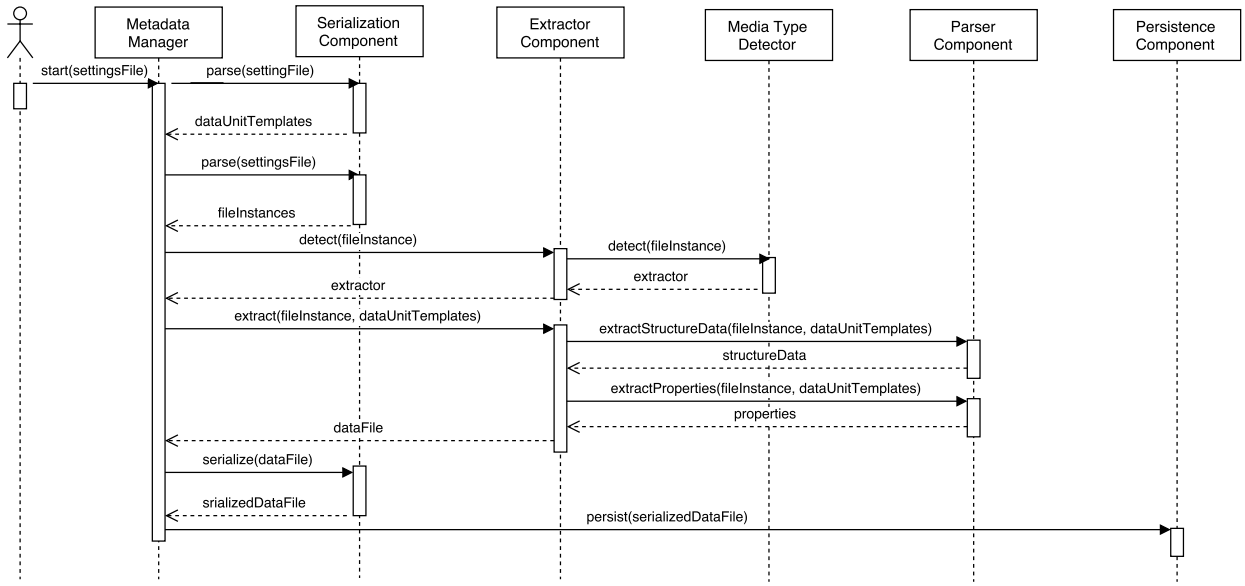

Figure 5. Behavioural overview

4.3 Tree Structure Inference Algorithm

Algorithm 1 represented in Figure 6 sketches the pruning procedure to detect the tree structure in a semi-structured data structure. The algorithm is less elaborate than similar approaches (e.g., [26]), but sufficient for our aim of supporting query formulation. It uses a Breadth-First-Search iteration over all the tree nodes to collect them in a stack. The leaf nodes are eventually at the top of the stack and the root – at the bottom. Then iterating this stack ensures that elements are visited from bottom to top. Each element e from the same level in the tree is pushed to a list *unless* there is an element there *equal* to e. If an equal element is contained in the list, the original one is removed form the tree. The list gets emptied on each change of the sub-tree root observed in the moment.

Informally, it runs bottom-to-top and starts by merging all the children of a node with the same name. In the simplified example (Figure 7), these are the two c leaf nodes. The procedure is repeated recursively level-wise to the top. Since both b nodes are on the same level and their children are the same (they both have a single c and a single e node), they are merged. The result is shown on the right part of the figure. The resulting representation is much more concise than the original tree and includes everything needed for future query procedures over the data it models.

As mentioned earlier, cardinality is included in each node, but is not shown on the picture for simplicity.

```
Data: stack bfsStack, list noDuplicates, breadth-first iterator bfIter
input: the root node root of a tree
Result: the tree rooted at root with all equivalent sub-trees pruned
prune()
    if root is empty then
        return
    end
    while bfIter has next do
        let child = bfIter.next()
        bfsStack.push(child)
    end
    /* root is now at the bottom, right-most leaf is at the top of
       the stack                                                    */

    let node = bfsStack.pop()
    while node.parent != null and bfsStack is not empty do
        let parent = node.parent
        /* traverse all nodes at the same level (node.parent == parent)
           */

        while node.parent == parent and bfsStack is not empty do
            if noDuplicates contains node then
                /* the tree rooted at node has equivalent tree which has
                   been visited, so remove the current one; note that
                   contains uses equals                               */

                remove node from tree
            else
                add node to noDuplicates
            end
            let node = bfsStack.pop()
        end
        empty noDuplicates
    end
```

Figure 6. Tree Structure Inference Algorithm (Algorithm 1)

4.4 System Implementation

As stated above, one important component in the implementation of the system is Apache Tika, which provides the basic framework to detect file types and to parse their content. However, as the structures of the files are different, various parsers are necessary to parse the internal data structures of a file. Figure 8 gives an overview of the implemented parsers.

The parser classes handle specific data formats and data file's schema instances and take care of both the metadata properties and the structure data. Of course, those two tasks are delegated to separate parsers for the same file type. This approach could be seen as a bit slower compared to one where a single parser extracts the metadata properties and infers the structure data. It is, however, much more extensible and flexible; on the other hand, the performance penalty is not that big either. Each specific parser knows the schema of the data it has to take care of. Similar to the Tika's approach, each parser's parse process is started through an interface common for all the parsers in the hierarchy (structure and metadata). Note that each parser knows about schema

specifics, but not concrete custom media types, so that a single parser could be reused in the future for other file types, having similar property or structure data arrangement. The StructureParser knows the type of metadata it has to parse and each of its derived classes knows how to parse its specific structure data. The same holds for the CustomMetadataParser and the derived types in its hierarchy.

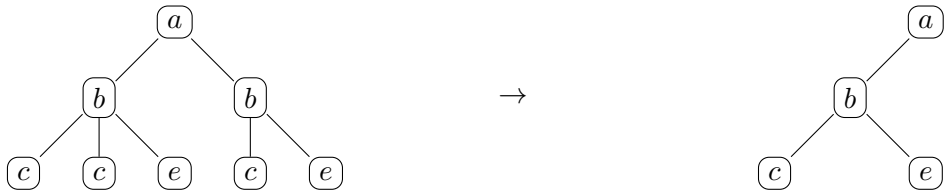

Figure 7. Example for the pruning procedure

A similar approach is applied for the implementation of the extractor components. For each specific data structure (e.g., CSV, XML), an extractor class is provided that implements a general interface. The extractor classes can be seen as a bridge between the parsers and the clients of those parsers. They hide the notion of different metadata, such as properties, structure, or semantic metadata and provide a single, unified interface to simply extract all known metadata in the best-effort manner from a given file. In addition, they also hide the parsers themselves, so the end-user only cares about different extractor types. For more convenient extensibility and maintainability, each extractor is associated with the media type of the file it is able to extract metadata from.

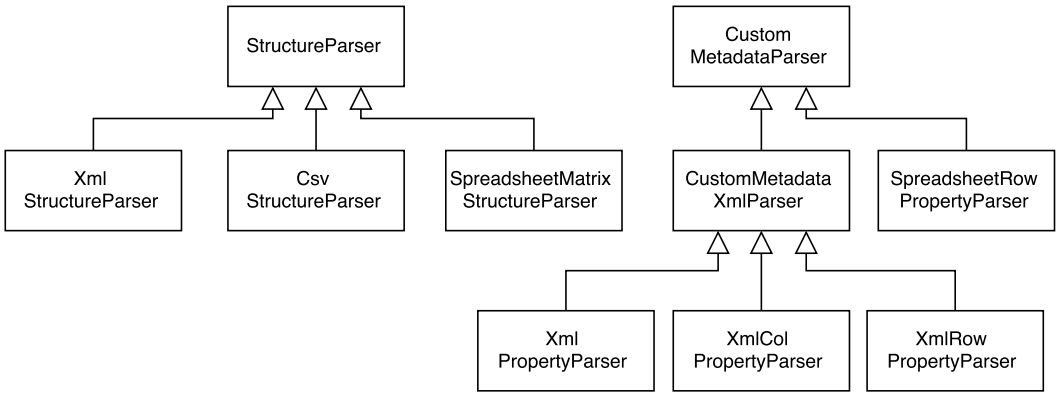

Figure 8. Parser Implementation

5 Evaluation

The goal of the evaluation is twofold. On the one hand, we want to show that GEMMS as a framework is actually useful and that it reduces the effort for metadata management in data lakes. Flexibility and extensibility were mentioned as the main requirements of the system, therefore, the first part of the evaluation (cf. Section 5.1) focuses on this aspect. On the other hand, we also evaluated the performance of the metadata extraction components, as it should be possible to apply the system to a large number of files (cf. Section 5.2).

As a programming framework, GEMMS is not intended to be used by end-users. An end-user application that uses GEMMS for metadata extraction is under development in the HUMIT project, but is yet not ready for evaluation.

5.1 Flexibility and Extensibility of GEMMS

GEMMS has been developed with standard file types (e.g., Excel, CSV, XML) and a few life-science-specific file types in mind. In the evaluation, we analyzed the required steps and efforts for the introduction of new file types. We used the file format Structured Data Format (SDF)[6] which is significantly different from the data formats considered earlier; the only commonality is that it is also text-based. The other file type is X-Chembench[7], which is very similar to CSV files. Both file types are used to describe chemical compounds or molecules.

The required steps are shown in the left column of Table 1. The 2nd and 3rd column indicate the number of lines of code to implement the required functionality.

The registration of the new media type (file type) is straightforward and just requires a few lines of XML code in the file custom-mimetypes.xml which is used by Tika to recognize file types (see Figure 9). The file types have also to be mentioned in one Java class. The definition of the data unit templates requires a little bit more work, as the structure and metadata properties of interest have to be defined. The most expensive part for SDF files is the parser, as these files have a specific structure that cannot be parsed by one of the existing parsers. For the Chembench file type, the existing CSV parser can be used. This would apply also to XML documents with a custom schema: the existing XML parser could be reused, the data unit template and the extractor just need to provide XPath expressions for the extraction of data units and metadata properties.

Table 1. Lines of code needed for each new file type

Step	SDF	Chembench	File	New
Media Type Registration	7	6	custom-mimetypes.xml	
	1	1	CustomTypes.java	
Data Unit Template	22	0	TabularTxtDataUnitTemplate.java	✓
	0	3	DataUnitTemplateDeserializer.java	
	1	1	DataUnitTemplates.java	
Parser	85	0	SdfPropertyParser.java	✓
Extractor	17	0	SdfExtractor.java	✓
	0	17	ChembenchDescriptorExtractor.java	✓
Mapping to Media Type	3	3	custom-mimetypes.xml	

After the parser has been defined, the extractor has to be implemented for the new file type. This component integrates the parser with the data unit templates and extracts the required data. Finally, the Tika configuration file has to be extended with a mapping of the file type to the new extractor class.

Overall, we can see that only very little efforts are required for the extension of the framework for new data file types. With an increasing number of file types already known by GEMMS, the effort for registering new file types should become smaller, as more code can be reused and the implementation of new parsers is not necessary. By the introduction of the new file types, it has further been shown that the designed metadata model is robust enough and there are no reasons to change it.

[6] SDF, https://en.wikipedia.org/wiki/Chemical_table_file#SDF.

[7] https://chembench.mml.unc.edu/help-fileformats

```
<mime-type type="chemical/x-mdl-sdfile">
    <sub-class-of type="text/plain"/>
    <acronym>sdf</acronym>
    <_comment>Structure-Data File</_comment>
    <glob pattern="*.sdf" />
    <glob pattern="*.mol" />
</mime-type>

<mime-type type="chemical/chembench-descriptor">
    <sub-class-of type="text/plain" />
    <acronym>x</acronym>
    <_comment>Descriptor file format used by Chembench</_comment>
    <glob pattern="*.x" />
</mime-type>

<mappings>
    <mapping type="chemical/x-mdl-sdfile">
        <mdms:extractor-class>de.fraunhofer.fit.mdms.extractor.SdfExtractor</mdms:extractor-class>
    </mapping>
    <mapping type="chemical/chembench-descriptor">
        <mdms:extractor-class>
                de.fraunhofer.fit.mdms.extractor.ChembenchDescriptorExtractor
        </mdms:extractor-class>
    </mapping>
</mappings>
```

Figure 9. An excerpt from the application configuration file: introduction of new media types and a mapping between them and extractor classes

5.2 Performance Measures

We evaluated the performance of three extractor classes that have been implemented during the development of GEMMS. The files are generated by hardware devices in our life science lab. The file types are proprietary file formats that have been specified by the manufacturers of these devices. The tests have been run for the three file types with distinction between metadata properties and structure data for two of the formats. Note that data formats do not correspond directly to the extension of the data file. For instance, a single format is based on spreadsheets and the other two are based on XML, but the data in them is structured differently. The first type of XML-based data format encodes its metadata and raw data in a straight-forward tree fashion. XML elements are nested in each other and metadata and raw data values are contained in the leaves. The second type of XML-based data format has a little more peculiar structure. It represents tables, in which the keys of metadata properties are all on the same row, while the values are in the corresponding cells on the row below. Raw data is contained in tables with header columns. Cells are child nodes of the row elements.

For each of the file types, the pair of structure and metadata properties parsers has been run three times in a row in a JUnit test method. The tests have been run on Java SE 1.8.0_45 on a Windows 7 Professional 64-bit Lenovo ThinkPad T440 with 8GB RAM and an Intel Core i5-4210U CPU at 1.7 GHz. The persistence layer of the application is realized with MongoDB 3.0.2. The mean of the those three run-time durations is what is shown in Table 2. The evaluation criteria in Table 2, *extraction* procedure is measured by the use of extractors plus file type detection and parsing of the ingestion process configuration string. During such an extractor run, an XML configuration string is parsed for the list of the input files and the data unit templates. To simulate real-world conditions, data unit templates for several different media types are encoded in the ingestion process configuration file used in the extraction procedures, even though a single data unit template is eventually matched. Just as with the parsing procedures, for each media type, the extraction procedure test case is run three times in a row and the mean run-time duration is considered.

As expected, the average durations of the parsing procedures are lower than the extraction procedures. The slower performance is caused by the parsing of the configuration string for the ingestion process and the automatic detection of the extractor suitable for the job. The great difference of both parsing and extraction times of the *application/x-nanodrop+xml* file type in

comparison to the other two, is caused by one particular file instance whose extraction and parsing has taken respectively 11.13 and 11.73 seconds on average.

Table 2. Performance of Parsers and Extractors

Media Type	File Count	Extraction Time (s)	Parsing Time (s)
x-2100bioanalyzer+xml	44	10.01	7.46
x-tecan+vnd.openxmlformats..	26	13.03	8.95
x-nanodrop+xml	27	25.83	24.07

6 Conclusions and Outlook

In this article, we proposed the generic and extensible metadata management system GEMMS, designed and implemented as the heart of a data lake, which should increase the productivity in analysis and management of heterogeneous data. Based on a classification of metadata - *structure data*, *metadata properties* and *semantic data* – we derived a generic, extensible and flexible metadata model providing easy accommodation for new or evolving metadata of various data sources. The framework is also extensible as new types of data sources can be easily integrated as we have shown in the evaluation.

Metadata extraction is one of the core features of our data lake implementation [27]. Based on the extracted metadata and schema information, further methods for the semantic enrichment of the data lake are currently being implemented. For instance, schema matching [28] is being applied to identify correspondences between the extracted metadata elements; schema summarization [19] creates consolidated representations of the extracted schemas; and mapping composition and query rewriting [17] are applied for data integration.

As our focus so far was on the extensibility of the core system, performance and user interfaces for end users require future work. Although performance is not a major concern (the ingestion process could be easily done as batch processes in regular intervals), a faster processing would be desirable. The querying functionality is yet simple (the user can query for data units annotated with certain ontology terms); an interactive query and exploration interface is one of the next milestones in the HUMIT project. Finally, we also need to consider database systems and similar systems as data sources; however, we are confident that the required changes or extensions will not break the core system which we have developed so far.

Acknowledgements. This work was supported by the German Federal Ministry of Education and Research (BMBF) under the project HUMIT (http://humit.de, FKZ 01IS14007A) and by the Klaus Tschira Stiftung under the mi-Mappa project (http://dbis.rwth-aachen.de/mi-Mappa/, project no. 00.263.2015).

References

[1] P. Vassiliadis and A. Simitsis, “Extraction, Transformation, and Loading,” in Encyclopedia of Database Systems, pp. 1095–1101, 2009. [Online]. Available: http://doi.org/10.1007/978-0-387-39940-9_158

[2] U. Dayal, M. Castellanos, A. Simitsis and K. Wilkinson, “Data Integration Flows for Business Intelligence,” in Proc. EDBT, pp. 1–11, 2009. [Online]. Available: https://doi.org/10.1145/1516360.1516362

[3] B. Stein and A. Morrison, “The Enterprise Data Lake: Better Integration and Deeper Analytics,” Technology Forecast: Rethinking integration, no. 1, pp. 1–9, 2014. [Online]. Available: http://www.pwc.com/us/en/technology-forecast/2014/cloud-computing/assets/pdf/pwc-technology-forecast-data-lakes.pdf

[4] I. Terrizzano, P.M. Schwarz, M. Roth and J.E. Colino, “Data Wrangling: The Challenging Journey From the Wild to the Lake,” in Proc. CIDR, pp. 9, 2015. [Online]. Available: http://www.cidrdb.org/cidr2015/Papers/CIDR15_Paper2.pdf

[5] S.C. Boulakia and U. Leser, “Next Generation Data Integration for Life Sciences,” in Proc. ICDE, pp. 1366–1369, 2011. [Online]. Available: http://doi.org/10.1109/ICDE.2011.5767957

[6] M.J. Villanueva, F. Valverde, A.M. Levin and O. Pastor, “Diagen: A Model-Driven Framework for Integrating Bioinformatic Tools,” in Selected Papers from CAiSE Forum 2011, ser. LNBIP, vol. 107, pp. 49–63, 2012. [Online]. Available: http://doi.org/10.1007/978-3-642-29749-6_4

[7] M. Stonebraker, D. Bruckner, I.F. Ilyas, G. Beskales, M. Cherniack, S.B. Zdonik, A. Pagan and S. Xu, “Data Curation at Scale: The Data Tamer System,” in Proc. 6th Conf. on Innovative Data Systems Research (CIDR), pp. 10, 2013. [Online]. Available: http://www.cidrdb.org/cidr2013/Papers/CIDR13_Paper28.pdf

[8] C. Quix, T. Berlage and M. Jarke, “Interactive Pay-As-You-Go-Integration of Life Science Data: The HUMIT Approach,” ERCIM News, no. 104, 2016. [Online]. Available: http://ercim-news.ercim.eu/en104/special/interactive-pay-as-you-go-integration-of-life-science-data-the-humit-approach

[9] M. Chessell, F. Scheepers, N. Nguyen, R. van Kessel and R. van der Starre, “Governing and Managing Big Data for Analytics and Decision Makers,” pp. 26, Aug. 2014. [Online]. Available: http://www.redbooks.ibm.com/redpapers/pdfs/redp5120.pdf

[10] A.Y. Halevy, F. Korn, N.F. Noy, C. Olston, N. Polyzotis, S. Roy and S.E. Whang, “Goods: Organizing Google’s Datasets,” in Proc. SIGMOD, pp. 795–806, 2016. [Online]. Available: https://doi.org/10.1145/2882903.2903730

[11] E. Boci and S. Thistlethwaite, “A Novel BIG DATA Architecture in Support of ADS-B Data Analytic,” in Proc. Integrated Communication, Navigation, and Surveillance Conference (ICNS), pp. C1–1–C1–8, April 2015. [Online]. Available: https://doi.org/10.1109/icnsurv.2015.7121281

[12] M. Franklin, A. Halevy and D. Maier, “From Databases to Dataspaces: A New Abstraction for Information Management,” SIGMOD Record, vol. 34, no. 4, pp. 27–33, 2005. [Online]. Available: https://doi.org/10.1145/1107499.1107502

[13] A.D. Sarma, X. Dong and A.Y. Halevy, “Bootstrapping Pay-As-You-Go Data Integration Systems,” in Proc. SIGMOD, pp. 861–874, 2008. [Online]. Available: https://doi.org/10.1145/1376616.1376702

[14] L. Blunschi, J.-P. Dittrich, O.R. Girard, S.K. Karakashian and M.A.V. Salles, “A Dataspace Odyssey: The Imemex Personal Dataspace Management System,” in Proc. CIDR, ser. CIDR ’07, pp. 114–119, 2007.

[15] P.A. Bernstein, A.Y. Halevy and R. Pottinger, “A Vision for Management of Complex Models,” SIGMOD Record, vol. 29, no. 4, pp. 55–63, 2000. [Online]. Available: https://doi.org/10.1145/369275.369289

[16] D. Kensche, C. Quix, M.A. Chatti and M. Jarke, “GeRoMe: A Generic Role Based Metamodel for Model Management,” Journal on Data Semantics, VIII, pp. 82–117, 2007. [Online]. Available: https://doi.org/10.1007/978-3-540-70664-9_4

[17] D. Kensche, C. Quix, X. Li, Y. Li and M. Jarke, “Generic Schema Mappings for Composition and Query Answering,” Data Knowl. Eng., vol. 68, no. 7, pp. 599–621, 2009. [Online]. Available: https://doi.org/10.1016/j.datak.2009.02.006

[18] E. Rahm and P.A. Bernstein, “A Survey of Approaches to Automatic Schema Matching,” VLDB Journal, vol. 10, no. 4, pp. 334–350, 2001. [Online]. Available: https://doi.org/10.1007/s007780100057

[19] C. Yu and H. Jagadish, “Schema Summarization,” in Proc. VLDB, Seoul, pp. 319–330, 2006.

[20] B. Smith, J.Köhler and A. Kumar, “On the Application of Formal Principles to Life Science Data: a Case Study in the Gene Ontology,” in 1st Intl. Workshop on Data Integration in the Life Sciences (DILS), ser. LNCS, vol. 2994. Leipzig, Germany: Springer, pp. 79–94, 2004. [Online]. Available: http://doi.org/10.1007/978-3-540-24745-6_6

[21] R.V. Guha, D. Brickley and S. Macbeth, “Schema.org: Evolution of Structured Data on the Web,” Commun. ACM, vol. 59, no. 2, pp. 44–51, 2016. [Online]. Available: http://doi.org/10.1145/2844544

[22] M. Klettke, U. Störl and S. Scherzinger, “Schema Extraction and Structural Outlier Detection for JSON-Based NoSQL Data Stores,” in Fachtagung Datenbanksysteme für Business, Technologie und Web (BTW), ser. LNI, T. Seidl, N. Ritter, H. Schöning, K. Sattler, T. Härder, S. Friedrich and W. Wingerath, Eds., vol. 241. Hamburg, Germany: GI, pp. 425–444, 2015. [Online]. Available: http://subs.emis.de/LNI/Proceedings/Proceedings241/article35.html

[23] J. Hegewald, F. Naumann and M. Weis, “Xstruct: Efficient Schema Extraction From Multiple and Large XML Documents,” in Proc. 22nd Intl. Conf. on Data Engineering (Workshops), IEEE, pp. 81–81, 2006. [Online]. Available: https://doi.org/10.1109/ICDEW.2006.166

[24] S. Abeyruwan, U.D. Vempati, H. Küçük-McGinty, U. Visser, A. Koleti, A. Mir, K. Sakurai, C. Chung, J.A. Bittker, P.A. Clemons, S. Brudz, A. Siripala, A.J. Morales, M. Romacker, D. Twomey, S. Bureeva, V.P. Lemmon and S.C. Schürer, “Evolving Bioassay Ontology (BAO): Modularization, Integration and Applications,” Journal of Biomedical Semantics, vol. 5, no. S-1, p. S5, 2014. [Online]. Available: https://doi.org/10.1186/2041-1480-5-s1-s5

[25] R.C. Martin, “Agile Software Development: Principles, Patterns and Practices,” Prentice Hall PTR, Upper Saddle River, NJ, 2003.

[26] G.J. Bex, F. Neven and S. Vansummeren, “Inferring XML Schema Definitions From XML Data,” in VLDB ’07 Proc. 33rd international conference on Very large data bases, Vienna, Austria, Sep. 23–27, pp. 998–1009, 2007. [Online]. Available: http://www.vldb.org/conf/2007/papers/research/p998-bex.pdf

[27] R. Hai, S. Geisler and C. Quix, “Constance: An Intelligent Data Lake System,” in Proc. SIGMOD, pp. 2097–2100, 2016. [Online]. Available: https://doi.org/10.1145/2882903.2899389

[28] C. Quix, D. Kensche and X. Li, “Matching of Ontologies with XML Schemas Using a Generic Metamodel,” in Proc. ODBASE, pp. 1081–1098, 2007. [Online]. Available: https://doi.org/10.1007/978-3-540-76848-7_71

[29] Proc. of SIGMOD/PODS’16 International Conference on Management of Data San Francisco, CA, USA, June 26–July 1, 2016. [Online]. Available: https://doi.org/10.1145/2882903