Article

Mapping hierarchical file structures to semantic data models for efficient data integration into research data management systems

Henrik Tom Wörden , Florian Spreckelsen , Stefan Luther , Ulrich Parlitz , Alexander Schlemmer *

Posted Date: 16 August 2023

doi: 10.20944/preprints202308.1170.v1

Keywords: research data management; FAIR; file structure; file crawler; semantic data model

Article

Mapping Hierarchical File Structures to Semantic Data Models for Efficient Data Integration into Research Data Management Systems

Henrik tom Wörden [4], Florian Spreckelsen [4], Stefan Luther [1,2,3,5], Ulrich Parlitz [1,2,3], Alexander Schlemmer [1,3,*]

[1] Max Planck Institute for Dynamics and Self-Organization, 37077 Göttingen, Germany
[2] Institute for the Dynamics of Complex Systems, Georg-August-Universität, 37077 Göttingen, Germany
[3] German Center for Cardiovascular Research (DZHK), partner site Göttingen, 37075 Göttingen, Germany
[4] Indiscale GmbH, 37083 Göttingen, Germany
[5] Institute of Pharmacology and Toxicology, University Medical Center Göttingen, 37075 Göttingen, Germany
[*] Correspondence: alexander.schlemmer@ds.mpg.de

Abstract: Although other methods exist to store and manage data in modern information technology, the standard solution are file systems. Therefore keeping well-organized file structures and file system layouts can be key to a sustainable research data management infrastructure. However, file structures alone are lacking several important capabilities for FAIR data management: The two most striking are insufficient visualization of data and inadequate possibilities for searching and getting an overview. Research data management systems (RDMS) can fill this gap, but many do not support the simultaneous use of the file system and the RDMS. This simultaneous use can have many benefits, but keeping data in the RDMS in synchrony with the file structure is challenging. Here, we present concepts that allow to keep file structures and semantic data models (in RDMS) synchronous. Furthermore, we propose a specification in yaml-format that allows for a structured and extensible declaration and implementation of a mapping between the file system and data models used in semantic research data management. Implementing these concepts will facilitate the re-use of specifications for multiple use cases. Furthermore, the specification can serve as a machine-readable and, at the same time, human-readable documentation of specific file system structures. We demonstrate our work using the Open Source RDMS CaosDB [1,2].

Keywords: research data management; FAIR; file structure; file crawler; semantic data model

1. Introduction

Data management for research is part of an active transformation which is required to meet the needs of increasing amounts of complex data. Furthermore, the FAIR guiding principles [3] for scientific data, which are an elementary part of numerous data management plans, funding guidelines and data management strategies of research organisations (e.g. [4,5]), require scientists to review and enhance their established data management workflows.

One particular focus of this endeavor is the introduction and expansion of research data management systems (RDMS). These systems help researchers to organize their data during the whole data management life-cycle, especially by increasing findability and accessibility [6]. Semantic data management approaches [7] furthermore can increase reuse and reproducibility of data that is typically organized in file structures. As it is pointed out in [6], one major shortcoming of file systems is the lack of rich metadata features which additionally limits the search possibilities.

Typically, RDMS employ database management systems (DBMS) to store data and meta data, but the degree to which data is migrated, linked or synchronized into these systems can highly vary. Concepts like Linked Data [8,9] or FAIR Digital Objects [10] provide overarching concepts for achieving more standardized representations within RDMS and for publication on the web.

Specific advantages of RDMS over organization of data in classical file hierarchies include:

- Advanced tagging and annotation
- Better comparability of data sets, possibly originating from different file formats and data representations
- Semantic information can be seamlessly integrated, possibly using standards like RDF [11] or OWL [12]
- Advanced querying and searching is possible, e.g. using SPARQL [13]

However, while acknowledging all these advantages, in this article we want to discuss the advantages of using classical file hierarchies complementary to RDMS and describe an approach to use both methods simultaneously. These concepts can be used independent of a specific RDMS software. However, as a proof-of-concept we have implemented the approach as part of the file crawler framework that belongs to the CaosDB [1,2] project. The crawler framework is released as Open Source software under AGPLv3 and can be accessed online[1]. This software is actively used in several projects, like [14].

1.1. *Using File Systems and RDMS Simultaneously*

Historically, the approach emerged within our project from requirements during the transition phase: While CaosDB was under development, we needed to incrementally synchronize files with our work-in-progress RDMS, making it subject to testing. Furthermore, we found it too risky to rely on a single system and wanted to be able to switch to our established fallback solution. Having reached a stable software product, we already had established our synchronization facilities and several additional advantages of using a normal folder structure simultaneous to our RDMS could be identified:

- Standard archiving techniques can be used for the file system securing data and meta data that is stored within files.
- Scientists can use their normal workflows to access and use files. An RDMS may be used in parallel with file access based workflows.
- Integration of various data acquisition and data analysis software is possible as the file system acts as a widely compatible interface.
- The file system can act as a fallback in cases where the RDMS might become unavailable. This methodology therefore increases robustness.
- Standard tools for managing the files can be used for backup (e.g. rsync), versioning (e.g. git) and file access (e.g. SSH). Functionality of these tools does not need to be re-implemented in the RDMS.

Especially the issue of interoperability with standard application programs has been already reported by [6]. For our proposed workflow it is highly beneficial, if a standardized file system layout like the one in [15] is used.

1.2. *Data Integration and Extract Transform Load (ETL)*

Every RDMS has to provide means to integrate data. While in some areas of electronic data processing, manual data insertion (e.g. using web forms) is still common, manual solutions are not feasible in scientific environments that typically handle data of large volume and complexity.

Automatic procedures for data integration in the context of data warehouses are typically known under the term ETL (Extract Transform Load) process [16,17]. These tools are responsible for extracting data from various, potentially heterogeneous sources, cleaning and customizing them and inserting the result into a data-warehouse or another data management system [18].

A typical ETL process can have a high complexity, possibly involving:

[1] https://gitlab.com/caosdb/caosdb-crawler

- Multiple sources of data and metadata
- Complex data transformations including conditions, loops and subprocesses
- Complex queries to MySQL or other databases

One standard way to notate and implement ETL processes is Busines Process Modeling Notation (BPMN) [19], a language standardized by the Object Management Group. This langauge can be used in tools like Kettle (Pentaho Data Integration) or Microsoft Integration Services for implementing complex ETL applications [17].

In practice, also custom data integration software written in various programming languages and making use of a high variety of different software packages, are used for data integration.

Writing ETL processes can be considered very costly [17]. In this article we are presenting a specialized concept for data integration for research data with a focus on semantic data management techniques. This concept is based on a simplified ETL process that is optimized to provide just enough flexibility to cover a wide range of possible file system and other data structures found in scientific data management. Furthermore, this concept involves methods for promoting standardization and proper documentation of research data.

1.3. Typical Structures of Research Data

We will illustrate the problem of integrating research data using a typical example which is based on the publication [15]. Although the concept is not restricted to data stored on file systems, we will assume for simplicity here, that the research data is stored on a standard file system with a well-defined file structure layout:

```
ExperimentalData/
  2020_SpeedOfLight/
    2020-01-01_TimeOfFlight
      README.md
      ...
    2020-01-02_Cavity
      README.md
      ...
    2020-01-03
      README.md
      ...
```

The above listing replicates an example with experimental data from [15] using a 3 level folder structure:

- Level 1 (`ExperimentalData`) stores rough categories for data, in this data acquired from experimental measurements.
- Level 2 (`2020_SpeedOfLight`) is the level of project names, grouping data into independent projects.
- Level 3 stores the actual measurement folders which can be also referred to as "scientific activity" folders in the general case. Each of these folders could have an arbitrary substructure and stores the actual experimental data along with a file README.md, containing meta data.

The typical use case of data integration for this example data involves the following sub tasks:

1. Identify the required data for integration into the RDMS. This can possibly involve information contained in the file structure (e.g. file names, path names or file extensions) or data contained in the contents of the files themselves.
2. Define an appropriate (semantic) data model for the desired data.

3. Specify the data integration procedure that maps data found on the file system (including data within the files) to the (semantic) data in the RDMS.

We discussed in Section 1.1 already that there are many use cases that can benefit from simultaneous usage of the file system and the RDMS. Therefore it is important to implement reliable means for identifying and transferring the data not only once, as a single "data ingest", but allowing for frequent updates of existing or changed data. Two common use cases for these kinds of updates are:

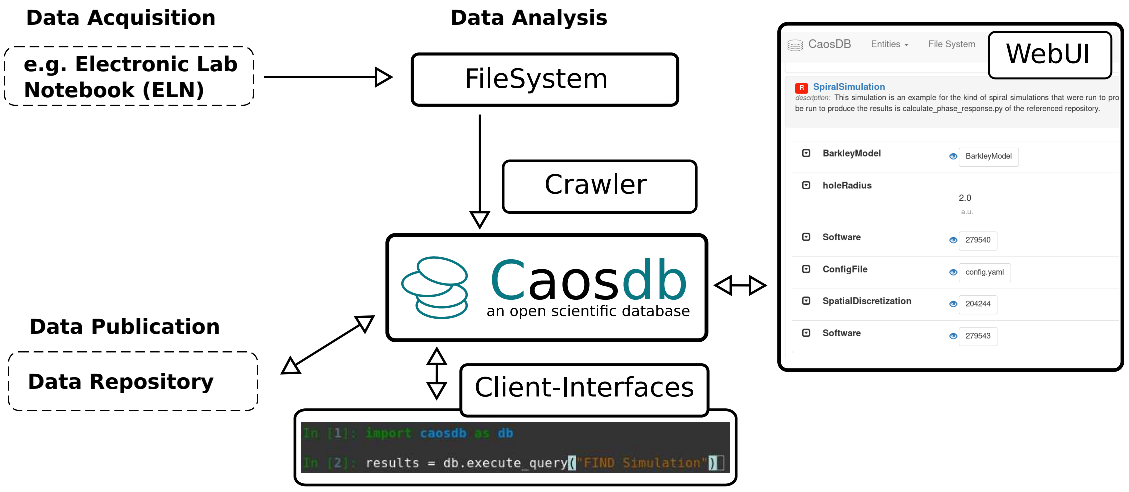
- An error in the raw data has been detected. It is corrected on the file system and the changes need to be propagated to the RDMS.
- Data files that are actively worked on have been inserted into the RDMS. A third-party software is used to process these files and consequently, the information taken from the files has to be frequently updated in the RDMS.

We are using the term "synchronization" here to capture the notion of allowing inserts of new and updates of existing data set in the same procedure. To avoid confusions, we want to explicitly note here that we don't refer to bi-directional synchronization. Bi-directional synchronization means that information from the RDMS that is not present in the file system can be propagated back to the file system which is not possible in our current implementation. Although ideas exist to implement bi-directional synchronization in the future, in the current work (and also the current software implementation) we focus on the uni-directional synchronization from the file system to the RDMS. Extensions to bi-directional synchronization will be discussed in the outlook in Section 3.2.2.

1.4. CaosDB

We use the semantic RDMS CaosDB [1,2] for demonstrating the concept. Figure 1 illustrates the role of CaosDB within the scientific data management workflows. CaosDB was designed as an RDMS mainly targeted at active data analysis. So, in contrast to electronic lab notebooks (ELNs), which have a stronger focus on data acquisition, and data repositories, which are used to publish data, data in CaosDB is assumed to be actively worked on by scientists on a regular basis. Its scope for single instances (which are usually operated on-premise) ranges from small work groups to whole research institutes. Independent of any RDMS, data acquisition typically leads to files stored on a file system. The CaosDB crawler synchronizes data from the file system into the RDMS. CaosDB provides multiple interfaces for interacting with the data, such as a graphical web interface and an API that can be used for interfacing the RDMS from multiple programming languages. CaosDB itself is typically not used as a data repository, but the structured and enriched data in CaosDB serves as a preparation for data publication and data can be exported from the system and published in data repositories. The semantic data model used by CaosDB is described in more detail in Section 2.2. CaosDB is Open Source software, released under AGPLv3 and can be found under https://gitlab.com/caosdb.

Figure 2 illustrates the modular design of the crawler: Data acquisition and other scientific activities can lead to heterogeneous file structures involving very different data formats. The crawler uses crawler specifications (CFoods) which define how these structures are interpreted and synchronized with CaosDB.



SpiralSimulation
description: This simulation is an example for the kind of spiral simulations that were run to pro
be run to produce the results is calculate_phase_response.py of the referenced repository.

BarkleyModel	BarkleyModel
holeRadius	2.0 a.u.
Software	279540
ConfigFile	config.yaml
SpatialDiscretization	204244
Software	279543

```
In [1]: import caosdb as db

In [2]: results = db.execute_query("FIND Simulation")
```

Figure 1. Overview over typical workflows involving CaosDB using the crawler. See Section 1.4 for details.. Figure was previously published [20].

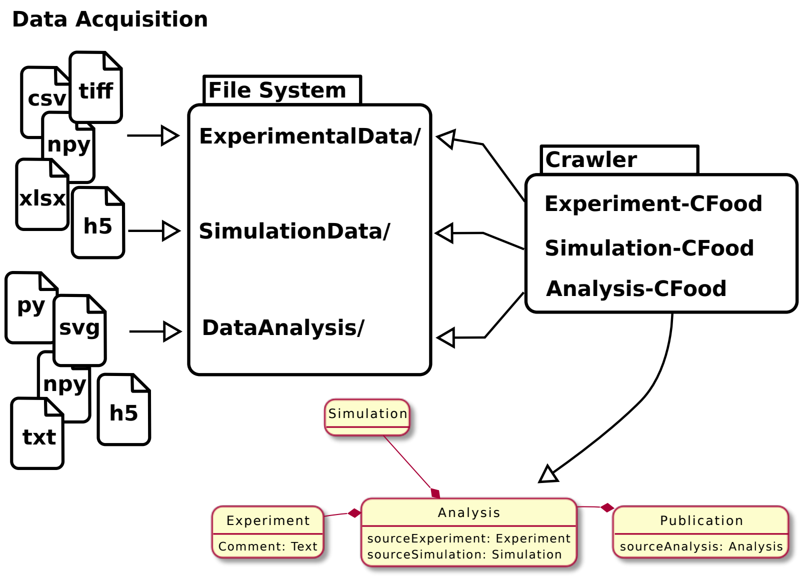


Figure 2. Illustration of the data synchronization procedure using a crawler: Data acquisition (possibly including computer simulations or data produced during data analysis) leads to a variety of files in different formats on the file system. Crawler-plugins ("CFoods") are designed in a way that they understand the local structures on the file system. They traverse the file tree, possibly opening files and extracting (meta) data in order to transform them into a semantic data model that is then synchronized with the RDMS as described in Section 2.4. Figure was previously published [21].

2. Results

Our procedure is composed of multiple elements:

- Semantic data model in the RDMS for the data to be integrated. We are using the CaosDB data model which is described briefly in Section 2.2 and in more detail in [1].
- Crawler definition with declarations of rules what parts of the file system or the file contents will be used, how they will get processed and how they are mapped onto the semantic data model. These crawler definitions (which we will refer to as `CFoods`) will be described in section Section 2.3.
- A specification of identity of objects which will allow us to check which objects are already present in the RDMS and therefore need an update instead of an insert operation. Our concept, which is

similar to unique keys in relational database management systems, will be discussed in Section 2.4.

Our implementation of the procedure, the CaosDB crawler, makes use of these three concepts in order to integrate data into the RDMS. Based on the example we introduced in Section 1.3 we will illustrate in the following sections how the information from the file system will be mapped onto semantic data in the RDMS.

2.1. Overview Over the Full Data Integration Procedure

Figure 3 gives an overview over the complete data integration procedure with the CaosDB crawler.

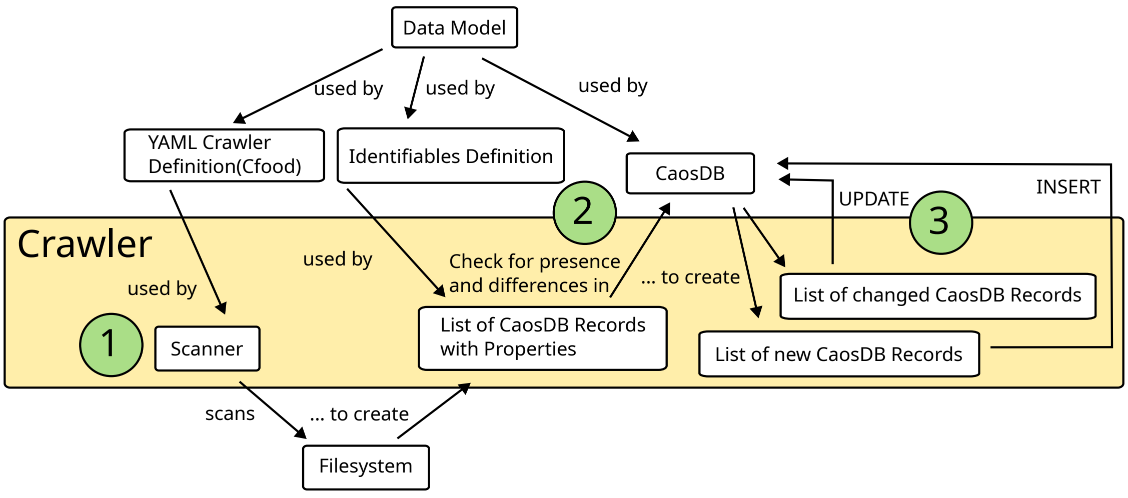

Data Model
used by
used by
used by
YAML Crawler Definition(Cfood)
Identifiables Definition
CaosDB
INSERT
UPDATE
2
3
Crawler
used by
used by
Check for presence and differences in
... to create
List of changed CaosDB Records
1
Scanner
List of CaosDB Records with Properties
List of new CaosDB Records
scans
... to create
Filesystem

Figure 3. Overview over the complete data integration procedure: In step (1), the scanner uses the YAML crawler definition (Section 2.3) to match converters to a given file system tree. From the information that is matched, a list of CaosDB Records with Properties is created. In step (2), the crawler checks which of these Records are already contained in CaosDB in order to separate the list of Records into a list of new Records and a list of changed Records. In order to complete this check, it makes use of a given definition of Identifiables (Section 2.4). In step (3) both lists are synchronized with the CaosDB server, i.e. all Records from the list of new Records are inserted into CaosDB and all Records from the list of changed Records are updated. The data model (Section 2.2) is needed to write a valid YAML crawler definition and to create the definition of Identifiables (Section 2.4). Furthermore, it is used by the CaosDB server directly.

There is a self-contained, documented example available online[2] that demonstrates the application of the crawler to example data.

2.2. Data Models in CaosDB

The CaosDB data model [1] is basically an object oriented representation of data which makes use of four different types of entities: `RecordType`, `Property`, `Record` and `File` `RecordTypes` and `Properties` define the data model which is later used to store concrete data objects which are represented by `Records`. In that respect, `RecordTypes` and `Properties` share a lot of similarities with ontologies, but have a restricted set of relations, as described in more detail in [1]. `Files` have a special role within CaosDB as they represent references to actual files on a file system, but allow to link them to other CaosDB entities and e.g. add custom properties.

`Properties` are individual pieces of information which have a name, description, optionally a physical unit and can store a value of a well defined data type. `Properties` are attached to `RecordTypes`

[2] https://gitlab.com/caosdb/documented-crawler-example

and can be marked as "obligatory", "recommended" or "suggested". In case of obligatory `Properties`, each `Record` of the respective `RecordType` is enforced to set the respective `Properties`. Each `Record` must have at least one `RecordType` and `RecordTypes` can have other `RecordTypes` as parents. This is known as (multiple) inheritance in object oriented programming languages.

In Figure 4 an example data model is shown in the right column in a UML-like diagram: There are three `RecordTypes` (`Project`, `Person`, `Experiment`), each having a small set of `Properties` (e.g. an integer `Property` called "year" or a `Property` referring to records of type `Person` called "responsible"). The red lines with the diamond show references between `RecordTypes`, i.e. where `RecordTypes` are used as `Properties` in other `RecordTypes`.

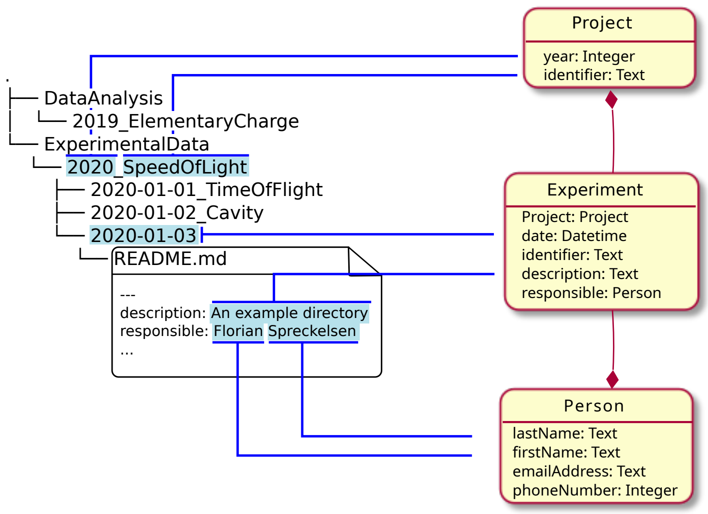


Figure 4. Mapping between file structure and data model. Blue lines indicate which pieces of information from the file structure and file contents are mapped to the respective properties in the data model.

More explanations and examples can be found online, e.g. in the official documentation of CaosDB[3].

2.3. Mapping Files and Layouts into a Data Model

Suppose we have a hierarchical structure of some kind that contains certain information and we want to map this information onto our object oriented data model. The typical example for the hierarchical structure would be a folder structure with files, but also hierarchical data formats like HDF5 [22,23] files (or a mixture of both) would fit the use case.

Figure 4 illustrates what such a mapping could look like in practice using the file structure introduced in Section 1.3:

- ExperimentalData contains one subfolder `2020_SpeedOfLight`, storing all data from this experimental series. The experimental series is represented in a `RecordType` called `Project`. The `Properties` "year" (2020) and "identifier" (SpeedOfLight) can be directly filled from the directory name.
- Each experiment of the series has its dedicated subfolder, so one `Record` of type `Experiment` will be created for each one. The association to its `Project` which is implictely clear in the folder hierarchy can be mapped to a reference to the `Project` `Record` created in the previous step. Again, the `Property` "date" can be set from the directory name.

[3] https://docs.indiscale.com

- Each experiment contains a file called `README.md` storing text and metadata about the experiment, according to the standard described in [15]. In this case the "description" `Property` for the `Experiment Record` created in the previous step will be set from the corresponding YAML value. Furthermore, a `Person Record` with `Properties` firstName = Florian and lastName = Spreckelsen will be created. Finally the `Person Record` will be set as value for the property "responsible" of the `Experiment Record`.

YAML Definitions

We created a special syntax in YAML format for implementing the steps which were just described in a machine-readable form. We call this description of rules for the crawler the crawler definition or `CFood`. For the example that was just described the corresponding YAML file would look like this:

```yaml
ExperimentalData_Dir:
  type: Directory
  match: "ExperimentalData"
  subtree:
    Project_Dir:
      type: Directory
      match: (?P<year>[0-9]{4,4})_(?P<name>.*)
      records:
        Project:
          year: $year
          name: $name
      subtree:
        Experiment_Dir:
          type: Directory
          match: (?P<date>[0-9]{4,4}_[0-9]{2,2}_[0-9]{2,2})(_(?P<identifier>.*))?
          records:
            Experiment:
              Project: $Project
              date: $date
              identifier: $identifier
          subtree:
            Readme_File:
              type: MarkdownFile
              match: "README.md"
              subtree:
                description:
                  type: DictTextElement
                  match_value: (?P<description>.*)
                  match_name: description
                  records:
                    Experiment:
                      description: $description
                responsible_single:
                    type: DictTextElement
                    match_name: responsible
                    match_value: ((?P<first_name>.+) )?(?P<last_name>.+)
                    records:
                      Person:
```

```
  first_name: $first_name
  last_name: $last_name
Experiment:
  Person: $Person
```

The YAML structure mimics a hierarchical structure of a file tree. The crawler operates by successively matching `Converters` to files, folders and possibly other `StructureElements`. We use the term `StructureElement` to specify any piece of information that is derived from some part of the file structure and that can be matched by a `Converter`. If a converter matches, `Records` are created corresponding to the entries under `records` in the crawler definition. Afterwards the crawler proceeds by processing sub-elements, like sub folders, using the converters defined under `subtree`.

In the given example the following converters can be found:

- `ExperimentalData_Dir`, `Project_Dir`, `Experiment_Dir`, all of type `Directory`. These converters match names of folders against the regular expression given by `match`. When matching, converters of this type yield sub-folders and -files for processing of converters in the section `subtree`.
- `Readme_File` is of type `MarkdownFile` and allows processing of the contents of the YAML header by converters given in the `subtree` section.
- `description` and `responsible_single` are two converters of type `DictTextElement` and can be used to match individual entries in the YAML header contained in the markdown file.

There are many more types of standard converters included in the CaosDB crawler. Examples include converters for interpreting tabular data (in Excel or CSV format), JSON [24,25] files or HDF5 [22,23] files. Custom converters can be created in Python by using the CaosDB crawler Python package. There is a community repository online[4] where community extensions are collected and maintained. Crawler extensions for prevalent scientific data structures like BIDS [26] and DICOM [27] can be found there.

Variables

Variables begin with a dollar sign and can be used in multiple occasions within the `CFood`, e.g. for setting properties of `Records`. They are created in multiple places, e.g.:

- From variables found in a matched regular expression. For example, `$year` is set from the `match` entry of the `Converter` `Project_Dir`.
- From record definitions. E.g. the `records` section of `Project_Dir` creates a variable `$Project` which will later be used as the value of a `Property` of `Record` `Experiment` created in the `records` section of `Experiment_Dir`.

Details about the syntax will be made available in the official documentation of the CaosDB crawler[5].

Scanner

We call the subroutines that gather information from the file system by applying the crawler definition to a file hierarchy the process of scanning. The corresponding module of the crawler software is called the `Scanner`. The result of a scanning process is a list of CaosDB `Records` where the values of the `Properties` are set to the information found.

In the example shown in Figure 4, the list will at least contain:

- Two `Project Records` with names "ElementaryCharge" and "SpeedOfLight"

[4] https://gitlab.com/caosdb/crawler-extensions
[5] https://docs.indiscale.com/caosdb-crawler/

- Three `Experiment Records` with dates "2020-01-01", "2020-01-02", "2020-01-03"
- One `Person Record` for "Florian Spreckelsen"

The remaining steps in the synchronization procedure are to split this list into a list of `Records` that need to be newly inserted and a list of`Records` that need an update. Sec.2.4 describes how the updates are distinguished from the insert operations. Sec.2.5 describes the final process of carrying out the transactions.

2.4. Identifiables

In order to determine which of the `Records` that have been generated during the scanning procedure in Section 4 is already present in the RDMS, the crawler needs information on how to determine identity of `Records`. For this purpose we implement a concept that is similar to unique keys which are used in the context of relational database management systems. For each `RecordType` we define a set of `Properties` that can be used to uniquely identify a single `Record`.

Using the example from Figure 4, we could claim that each `Person` in our RDMS can be uniquely identified by giving a "firstName" and a "lastName". It is important to point out that the definition of identities for `RecordTypes` can highly vary depending on the usage scenario and environment: There might of course be many cases where this example definition is not sufficient, because persons can have the same first and last names.

In this case we would define the identity of `RecordType Person` by declaring that the set of the properties "firstName" and "lastName" is the `Registered Identifiable`[6] for `RecordType Person`. During scanning, `Identifiables` are filled with the necessary information. In our example from Figure 4, "firstName" will be set to "Florian" and "lastName" to "Spreckelsen". We refer to this entity as the `Identifiable`. Subsequently, this entity is used to check whether a `Record` with these property values already exists in CaosDB. If there is no such `Record`, a new `Record` is inserted. This `Record` can of course have much more information in the form of properties attached to it, like an "emailAddress", which is not part of the `Identifiable`. If such a `Record` already exists in CaosDB, it is retrieved. We refer to the retrieved entity as `Identified Record`. This entity will then be updated as described in Section 2.5. The terminology, which we introduce here, is summarized in Figure 5.

To summarize the concept of `Identifiables`:

- An `Identifiable` is a set of properties that is taken from the file system layout or the file contents that allow to uniquely identify the corresponding object in the RDMS
- The set of properties is tied to a specific `RecordType`
- They may or may not contain references to other objects within the RDMS
- `Records` can of course contain much more information than is contained in the `Identifiable`. This information is stored in other `Properties`, which are not part of the `Identifiable`.

[6] We use the term "Identifiable" here, because we want to avoid confusions with the concept of "unique keys" which share some similarities, but are also different in some important aspects. Furthermore, the RDMS CaosDB also makes use of unique keys as part of its usage of a relational database management system in its backend.

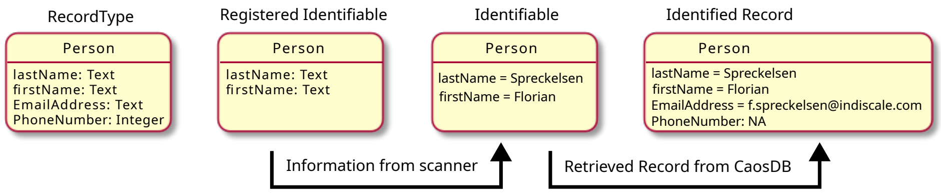


Figure 5. Terminology used in the context of defining identity for `Records`: For each `RecordType` that is used in the synchronization procedure its identity needs to be defined in what we call the `Registered Identifiable`. The scanner will fill the values of the properties of the `Registered Identifiable` and in that process create an `Identifiable`. This `Identifiable` can be used by the crawler to run a query on CaosDB. If a `Record` matching the properties of the `Identifiable` already exists in CaosDB, it will be retrieved and used as described in Section 2.5 to update the `Record`. Otherwise a new `Record` will be inserted.

Ambigiously Defined `Identifiables`

Checking the existence of an Identifiable in CaosDB is done using a query. If this query returns either zero or exactly one entity, `Identifiables` can be uniquely identified. In case the procedure returns two or more entities, this indicates that the `Identifiable` is not designed properly and that the given information is not sufficient to uniquely identify an entity. One example might be that there are in fact two persons with the same "firstName" and "lastName" in the RDMS. In this case the `Registered Identifiable` needs to be adapted. The next paragraph discusses the design of `Identifiables`.

Proper Design of `Identifiables`

The general rule for designing `Identifiables` should be to register the minimum amount of information that needs to be checked to uniquely identify a `Record` of the corresponding type. If too much (unneeded) information is included, it is no longer possible to update parts of the `Record` from information from the file system, as small changes will already lead to a `Record` with a different identity. If, on the other hand, too little information is included in the `Identifiable`, the `Records` might be ambiguously defined, or might become ambiguous in the future. This issue is discussed in the previous paragraph.

2.5. Inserts and Updates

The final step in the crawling procedure is the propagation of the actual transactions into the RDMS. This is illustrated in Figure 3 as step three. As described in Section 2.4, `Identifiables` can be used to separate the list of `Records`, which were created by the scanner in step one, into a list of new `Records` and a list of changed `Records`. The list of new `Records` can just be inserted into the RDMS. For the list of changed `Records`, each entity is retrieved from the server first. Each of these `Records` is then overwritten with the new version of the `Record`, as it was generated by the scanner. Afterwards the `Records` are updated in the RDMS.

3. Discussion

3.1. Benefits

In the previous sections we presented a data integration concept and a software framework implementing it. The main goal was to facilitate scientific data management workflows that benefit from a simultaneous usage of file systems and semantic data management. In this section we discuss some important design decisions, benefits from the approach and limitations.

YAML Specifications as Abstraction Layer for Data Integration

In Sec.4 our approach to map data found on the file system to a semantic data model was presented along with a formalized syntax in YAML. This syntax was designed to be an abstraction of common data integration tasks, like matching directories, files and their contents, but still be flexible enough to allow for the integration of very complex data or very rare file formats.

One advantage of using an abstraction layer that is a middle way between very specific source code for data integration and highly standardized routines is, that crawler specifications can be re-used in a greater variety of scenarios. Because the hierarchical structure of the YAML crawler specification corresponds to the hierarchical structure of file systems and file contents, it is much simpler to identify similarities between different file structures, than it would be with plain data integration source code. Furthermore, the crawler specifications are machine-readable and therefore open the possibility for much more complex applications.

Primary Source of Information

Our update procedure synchronizes data from the file system into the RDMS uni-directionally, so we can say that the file system is the single source of truth.

Another possibility would have been to implement merges between entities generated from file system information and entities present in the RDMS. This would have allowed to edit entities simultaneously on the file system and in the RDMS and obtain a merged version after the crawling procedure. However, we decided against this and chose a single source of information approach. In practice, merges can become really complicated and having a clearly defined source of information makes the procedure much more transparent and predictable to the users. Furthermore, we found, that use cases involving simultaneous edits of entities on the file system and in the RDMS are rather rare, so we decided against adding this additional complexity into our software.

We consider as best practice to not edit entities generated by the crawler in the RDMS directly, but use references to these entities instead. This can be enforced by CaosDB by automatically setting the entities generated by the crawler as read-only for other users. CaosDB entities are protected against accidental data-loss using versioning of entities.

Documentation of Data Structures and File Hierarchies

The approach we presented here purposefully relies on a manual definition of the semantic data model and a careful definition of the synchronization rules. Sometimes it is criticized that this can involve a lot of work. A frequent suggestion is to rather apply machine learning techniques / artificial intelligence methods to organize data and make it more findable. However, it is important to highlight that the manual design and documentation of file structures and file hierarchies actually has several beneficial side-effects:

- One of the main goals of managing and organizing data is to enable researchers to better understand, find and re-use their data [3]. A semantic data model created by hand captures an understanding of researchers of their data. Data organized by an artificial intelligence is very likely not represented in a way that is easily understood by the researcher and does not incorporate the same meaning.
- The process of designing the data models and rules for synchronization is a creative process that leads to optimized research workflows. In our experience, researchers highly benefit from the process of structuring their own data management.
- Many machine learning methods rely on large amounts of training data for being accurate and efficient. Often these data are not available and therefore a manual step that is likely to be equally time-consuming to the design of the data model and synchronization rules is necessary.

As a practical outcome, the data documentation created using this procedure can be used to create data management plans, which are nowadays an important requirement for institutions and funding agencies. Although we think, that the design of the process should be in parts manually, integrating artificial intelligence in the form of assistants is possible. An example for such an assistant could be an algorithm that generates a suggested crawler definition based on existing data which then could be corrected and expanded by the researcher.

3.2. Limitations

3.2.1. Deletion of Files

One important limitation of the approach presented here, is that deletions on the file system are not directly mapped to the RDMS, i.e. the records stemming from deleted files will persist in the RDMS and have to be deleted manually. This design decision was intended as it allows for complex distributed workflows. One example is that two researchers from the same work group work on two different projects with independent file structures. Using our approach it is possible to run the crawler on two independent file trees and thereby update different parts of the RDMS, without having to synchronize the file systems before.

Future implementations of the software could make use of file system monitoring to implement proper detection of deleted files. Another possibility would be to signal deletion of files and folders by special files (e.g. special names or contents) that trigger an automatic deletion by the crawler.

3.2.2. Bi-Directional Synchronization

A natural extension of the concepts presented here would be a crawler that allows for bi-directional synchronization. In addition to inserts and updates that are propagated from the file structure to the RDMS, changes in the RDMS would also be detected and propagated to the file system leading the creation and updating of files. While some parts of these procedure, like identifying changes in the RDMS, can be implemented in a straight-forward way, the mapping of information to existing file trees can be considered quite complex and raises several questions. While the software in its current form needs only read-only access to the file system, in a bi-directional scenario read-write-access is required, so that more care has to be taken to protect the users from unwanted data-loss.

4. Conclusions

In this article we have presented a structured approach for data integration from file systems into RDMS. We have shown, how this concept is applied practically and we have published an Open Source software framework as one implementation of this concept. In multiple active data management projects we have found that this mixture of standardized definition of the data integration with the possibility to extend them with flexible custom code allows for a rapid development of data integration tools and facilitates re-use of data integration modules.

Author Contributions: Conceptualization, A.S., H.t.W. and F.S.; software, A.S., H.t.W. and F.S.; resources, A.S., H.t.W., U.P., S.L. and F.S.; writing—original draft preparation, A.S., H.t.W. and F.S.; writing—review and editing, A.S., H.t.W., U.P., S.L. and F.S.; supervision, A.S. and H.t.W.; project administration, H.t.W. and S.L.; funding acquisition, A.S., U.P. and S.L. All authors have read and agreed to the published version of the manuscript.

Funding: A.S., U.P. and S.L. acknowledge financial support by the Volkswagen Stiftung within the framework of the project: "Global Carbon Cycling and Complex Molecular Patterns in Aquatic Systems: Integrated Analyses Powered by Semantic Data Management". S.L. acknoweldges financial support by the DZHK and DFG SFB 1002 Modulary Units in Heart Failure, and the Else Kröner-Fresenius Foundation (EKFS).

Conflicts of Interest: A.S. and H.t.W. are co-founders of IndiScale GmbH, a company providing commercial services for CaosDB. Furthermore, H.t.W. and F.S. are employees of IndiScale GmbH. The funders had no role in the design of the study; in the collection, analyses, or interpretation of data; in the writing of the manuscript, or in the decision to publish the results.

Abbreviations

The following abbreviations are used in this manuscript:

BIDS	Brain Imaging Data Structure
CQL	CaosDB Query Language
CSV	Comma-Separated Values
DICOM	Digital Imaging and Communications in Medicine
ELN	Electronic Lab Notebook
ETL	Extract Transform Load
FAIR	Findable, Accessible, Interoperable and Reusable
FDO	FAIR Digital Object
HDF5	Hierarchical Data Format
JSON	JavaScript Object Notation
LD	Linked Data
md	markdown
OWL	Web Ontology Language
RDF	Resource Description Framework
RDMS	Research-data management system
SPARQL	SPARQL Protocol and RDF Query Language
SQL	Structured Query Language
SSH	Secure Shell
YAML	YAML Ain't Markup Language

Appendix A. Supporting Software

The following software projects can be used to implement the workflows described in the article:

- Repository of the CaosDB-Open Source project: https://gitlab.com/caosdb
- Repository of the CaosDB-Crawler: https://gitlab.com/caosdb/caosdb-crawler

References

1. Fitschen, T.; Schlemmer, A.; Hornung, D.; tom Wörden, H.; Parlitz, U.; Luther, S. CaosDB—Research Data Management for Complex, Changing, and Automated Research Workflows. *Data* **2019**, *4*, 83. doi:10.3390/data4020083.
2. Hornung, D.; Spreckelsen, F.; Weiß, T. Agile Research Data Management with Open Source: CaosDB **2023**.
3. Wilkinson, M.D.; Dumontier, M.; Aalbersberg, I.J.; Appleton, G.; Axton, M.; Baak, A.; Blomberg, N.; Boiten, J.W.; da Silva Santos, L.B.; Bourne, P.E.; Bouwman, J.; Brookes, A.J.; Clark, T.; Crosas, M.; Dillo, I.; Dumon, O.; Edmunds, S.; Evelo, C.T.; Finkers, R.; Gonzalez-Beltran, A.; Gray, A.J.; Groth, P.; Goble, C.; Grethe, J.S.; Heringa, J.; 't Hoen, P.A.; Hooft, R.; Kuhn, T.; Kok, R.; Kok, J.; Lusher, S.J.; Martone, M.E.; Mons, A.; Packer, A.L.; Persson, B.; Rocca-Serra, P.; Roos, M.; van Schaik, R.; Sansone, S.A.; Schultes, E.; Sengstag, T.; Slater, T.; Strawn, G.; Swertz, M.A.; Thompson, M.; van der Lei, J.; van Mulligen, E.; Velterop, J.; Waagmeester, A.; Wittenburg, P.; Wolstencroft, K.; Zhao, J.; Mons, B. The FAIR Guiding Principles for scientific data management and stewardship. *Scientific Data* **2016**, *3*, 160018. doi:10.1038/sdata.2016.18.
4. Deutsche Forschungsgemeinschaft. Guidelines for Safeguarding Good Research Practice. Code of Conduct, 2022. Available in German and in English., doi:10.5281/zenodo.6472827.
5. Ferguson, L.M.; Bertelmann, R.; Bruch, C.; Messerschmidt, R.; Pampel, H.; Schrader, A.C.; Schultze-Motel, P.; Weisweiler, N.L. Good (Digital) Research Practice and Open Science Support and Best Practices for Implementing the DFG Code of Conduct "Guidelines for Safeguarding Good Research Practice". Helmholtz Open Science Briefing. Version 2.0 **2022**.
6. Gray, J.; Liu, D.T.; Nieto-Santisteban, M.; Szalay, A.; DeWitt, D.J.; Heber, G. Scientific data management in the coming decade. *34*, 34–41. doi:10.1145/1107499.1107503.
7. Samuel, S. Integrative Data Management for Reproducibility of Microscopy Experiments. The Semantic Web; Blomqvist, E.; Maynard, D.; Gangemi, A.; Hoekstra, R.; Hitzler, P.; Hartig, O., Eds.; Springer International Publishing: Cham, 2017; pp. 246–255.

8. Bizer, C.; Heath, T.; Ayers, D.; Raimond, Y. Interlinking Open Data on the Web. p. 2.
9. Bizer, C.; Heath, T.; Berners-Lee, T. Linked Data - The Story So Far. p. 26.
10. De Smedt, K.; Koureas, D.; Wittenburg, P. FAIR Digital Objects for Science: From Data Pieces to Actionable Knowledge Units. *8*, 21. doi:10.3390/publications8020021.
11. McBride, B. The resource description framework (RDF) and its vocabulary description language RDFS. In *Handbook on ontologies*; Springer, 2004; pp. 51–65.
12. OWL 2 Web Ontology Language Document Overview (Second Edition). p. 7.
13. Pérez, J.; Arenas, M.; Gutierrez, C. Semantics and Complexity of SPARQL. *ACM Trans. Database Syst.* **2009**, *34*. doi:10.1145/1567274.1567278.
14. Schlemmer, A.; Lennartz, S. Transparent and reproducible data analysis workflows in Earth System Modelling combining interactive notebooks and semantic data management. Technical report, Copernicus Meetings, 2023.
15. Spreckelsen, F.; Rüchardt, B.; Lebert, J.; Luther, S.; Parlitz, U.; Schlemmer, A. Guidelines for a Standardized Filesystem Layout for Scientific Data. *Data* **2020**, *5*, 43. doi:10.3390/data5020043.
16. Vassiliadis, P. A Survey of Extract–Transform– Load Technology **2009**. p. 75.
17. Vaisman, A.; Zimányi, E. *Data Warehouse Systems*; Springer Berlin Heidelberg. doi:10.1007/978-3-642-54655-6.
18. Vassiliadis, P.; Simitsis, A.; Skiadopoulos, S. Conceptual Modeling for ETL Processes. Proceedings of the 5th ACM International Workshop on Data Warehousing and OLAP; Association for Computing Machinery: New York, NY, USA, 2002; DOLAP '02, p. 14–21. doi:10.1145/583890.583893.
19. Business Process Model and Notation (BPMN), Version 2.0. p. 538.
20. Schlemmer, A. Data Management Workflows with CaosDB, 2021. doi:10.5281/zenodo.8246587.
21. Schlemmer, A. Mapping data files to semantic data models using the CaosDB crawler, 2021. doi:10.5281/zenodo.8246645.
22. Koranne, S., Hierarchical Data Format 5 : HDF5. In *Handbook of Open Source Tools*; Springer US: Boston, MA, 2011; pp. 191–200. doi:10.1007/978-1-4419-7719-9_10.
23. Folk, M.; Heber, G.; Koziol, Q.; Pourmal, E.; Robinson, D. An overview of the HDF5 technology suite and its applications. Proceedings of the EDBT/ICDT 2011 workshop on array databases, 2011, pp. 36–47.
24. Pezoa, F.; Reutter, J.L.; Suarez, F.; Ugarte, M.; Vrgoč, D. Foundations of JSON schema. Proceedings of the 25th international conference on World Wide Web, 2016, pp. 263–273.
25. Bray, T. The javascript object notation (json) data interchange format. Technical report, 2014.
26. Gorgolewski, K.J.; Auer, T.; Calhoun, V.D.; Craddock, R.C.; Das, S.; Duff, E.P.; Flandin, G.; Ghosh, S.S.; Glatard, T.; Halchenko, Y.O.; others. The brain imaging data structure, a format for organizing and describing outputs of neuroimaging experiments. *Scientific data* **2016**, *3*, 1–9.
27. Mildenberger, P.; Eichelberg, M.; Martin, E. Introduction to the DICOM standard. *European radiology* **2002**, *12*, 920–927.