Article

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

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

1 Indiscale GmbH, 37083 Göttingen, Germany; h.tomwoerden@indiscale.com (H.t.W.); f.spreckelsen@indiscale.com (F.S.)
2 Max Planck Institute for Dynamics and Self-Organization, 37077 Göttingen, Germany; stefan.luther@ds.mpg.de (S.L.); ulrich.parlitz@ds.mpg.de (U.P.)
3 Institute for the Dynamics of Complex Systems, Georg-August-Universität, 37077 Göttingen, Germany
4 German Center for Cardiovascular Research (DZHK), Partner Site Göttingen, 37075 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 is 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 lack several important capabilities for FAIR data management: the two most significant being insufficient visualization of data and inadequate possibilities for searching and obtaining an overview. Research data management systems (RDMSs) can fill this gap, but many do not support the simultaneous use of the file system and RDMS. This simultaneous use can have many benefits, but keeping data in RDMS in synchrony with the file structure is challenging. Here, we present concepts that allow for keeping 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 LinkAhead (previously named "CaosDB").

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

Citation: tom Wörden, H.; Spreckelsen, F.; Luther, S.; Parlitz, U.; Schlemmer, A. Mapping Hierarchical File Structures to Semantic Data Models for Efficient Data Integration into Research Data Management Systems. *Data* **2024**, *9*, 24. https://doi.org/10.3390/data9020024

Academic Editor: Kamran Sedig

Received: 15 August 2023
Revised: 8 December 2023
Accepted: 18 December 2023
Published: 26 January 2024

Copyright: © 2024 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (https://creativecommons.org/licenses/by/4.0/).

1. Introduction

Data management for research is part of an active transformation, which is required in order meet the needs of increasing amounts of complex data. Furthermore, the FAIR guiding principles [1] for scientific data, which are an elementary part of numerous data management plans, funding guidelines and data management strategies of research organisations (e.g., [2,3]), 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 (RDMSs). These systems help researchers organize their data during the whole data management life cycle, especially by increasing findability and accessibility [4]. Furthermore, semantic data management approaches [5] can increase the reuse and reproducibility of data that are typically organized in file structures. As it was pointed out in [4], one major shortcoming of file systems is the lack of rich metadata features, which additionally limits search options. Typically, RDMSs employ database

management systems (DBMSs) to store data and meta data, but the degree to which data is migrated, linked, or synchronized into these systems can vary substantially.

The import of data into an RDMS typically requires the development of data integration procedures that are tied to the specific workflows at hand. While very few standard products exist [6], in practice, mostly custom software written in various programming languages and making use of a high variety of different software packages are used for data integration in scientific environments. There are two main workflows for integrating data into RDMSs: Manual input of data (e.g., using forms [7]) or facilities for the batch import of data sets. The automatic methods often include data import routines for predefined formats, like tables in Excel or CSV format [8,9]. Some systems include plugin systems to allow for a configuration of the data integration process [10]. Sometimes, data files have to be uploaded using a web front-end [11] and are afterwards attached to objects in the RDMSs. In general, developing this kind of software can be considered very costly [6], as it is highly dependent on the specific environment. Data import can still be considered one of the major bottlenecks for the adaption of an RDMS.

There are several advantages for using an RDMS over organization of data in classical file hierarchies. There is a higher flexibility in adding metadata to data sets, while these capabilities are limited for classical file systems. The standardized representation in an RDMS improves the comparability of data sets that possibly originate from different file formats and data representations. Furthermore, semantic information can be seamlessly integrated, possibly using standards like RDF [12] and OWL [13]. The semantic information allows for advanced querying and searching, e.g., using SPARQL [14]. Concepts like Linked Data [15,16] and FAIR Digital Objects (FDO [17]) provide overarching concepts for achieving more standardized representations within RDMSs and for publication on the web. Specifically, the FDO concept aims at bundling data sets with a persistent digital identifier (PID) and its meta data to self-contained units. These units are designed to be machine-actionable and interoperable, so that they have the potential to build complex and distributed data processing infrastructures [17].

1.1. Using File Systems and RDMSs Simultaneously

Despite the advantages mentioned above, RDMSs have still failed to gain a widespread adoption. One of the key problems in the employment of an RDMS in an active research environment is that a full transition to such a system is very difficult: Most digital scientific workflows are in one or multiple ways dependent on classical hierarchical file systems [4]. Examples include data acquisition and measurement devices, data processing and analysis software, and digitized lab notes and material for publications. The complete transition to an RDMS would require developing data integration procedures (e.g., extract transform load (ETL) [6,18] processes) for every digital workflow in the lab and to provide interfaces for in- and output to any other software involved in these workflows.

As files on classical file systems play a crucial role in these workflows, our aim is to develop a robust strategy to use file systems and an RDMS simultaneously. Rather than requiring a full transition to an RDMS, we want to make use of the file system as an interoperability layer between the RDMS and any other file-based workflow in the research environment.

There are two important tasks that need to be solved and that are the main focus of this article:

1. There must be a method to keep data and meta data in the RDMS synchronized with data files on the file system. Using that method, the file system can be used as interoperability layer between the RDMS and other software and workflows. Our approach to solving this issue is discussed in detail in Section 2.1. One key component of the synchronization method is the definition of identity for data in the RDMS, which is discussed in Section 2.3.
2. The high variety of different data structures found on the file system needs an adaptive and flexible approach for data integration and synchronization into the RDMS. We

discuss our solution for this task in Section 2.2.1 where we present a standardized but highly configurable format for mapping information from files to a semantic data model.

Apart from the main motivation, described above, we have identified several additional advantages of using a conventional folder structure simultaneous to an RDMS: Standard tools for managing the files can be used for backup (e.g., rsync), versioning (e.g., git), archiving, and file access (e.g., SSH). Functionality of these tools does not need to be re-implemented in the RDMS. Furthermore, the file system can act as a fallback in cases where the RDMS might become unavailable. This methodology, therefore, increases robustness. As a third advantage, existing workflows relying on storing files in a file system do not need to be changed, while the simultaneous findability within an RDMS is available to users.

The concepts described in this article can be used independent of a specific RDMS software. However, as a proof-of-concept, we implemented the approach as part of the file crawler framework that belongs to the LinkAhead [19,20] project (CaosDB was recently renamed LinkAhead.). The crawler framework is released as Open Source software under AGPLv3 (see Appendix A).

1.2. Example Data Set

We will illustrate the problem of integrating research data using a simplified example that is based on the publication of [21]. This example will be used in Section 2 to demonstrate our data integration concepts. Examples for more complex data integration, e.g., for data sets found in the neurosciences (BIDS [22] and DICOM [23]) and in the geosciences, can be found online (see Appendix B). Although the concept is not restricted to data stored on file systems, we will assume for simplicity here, that the research data are 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 [21] using a three-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 also be referred to as "scientific activity" folders in the general case. Each of these folders could have an arbitrary substructure and store the actual experimental data along with a file README.md, containing meta data.

The generic use case of integrating data from file systems 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.

A concrete example for this procedure including a semantic data model is provided in Section 2.2. We discuss in Section 1.1 that there are already many use cases that can benefit from the simultaneous use of the file system and RDMS. Therefore, it is important to implement reliable means for identifying and transferring the data not only once, as a single "data import", but allowing for frequent updates of existing or changed data. Such an update might be needed if an error in the raw data has been detected. It can then be corrected on the file system and the changes need to be propagated to the RDMS. Another possibility is that 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 use the term "synchronization" here to refer to the possible insertion of new data sets and to update existing data sets in the same procedure. To avoid confusion, we want to explicitly note here that we are not referring to bi-directional synchronization. Bi-directional synchronization means that information from 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 3.4.2.

1.3. LinkAhead

In this section, LinkAhead and its data model will be briefly introduced, as this software will be used for demonstrating our data integration concept. LinkAhead 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 LinkAhead are assumed to be actively worked on by scientists on a regular basis. Its scope for single instances (which are usually operated on-premises) 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 LinkAhead crawler synchronizes data from the file system into the RDMS. LinkAhead 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. LinkAhead itself is typically not used as a data repository, but the structured and enriched data in LinkAhead 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 LinkAhead is described in more detail in Section 1.3.1. LinkAhead is an open source software, released under AGPLv3 (see Appendix A).

1.3.1. Data Models in LinkAhead

The LinkAhead data model 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 [19]. `Files` have a special role within LinkAhead as they represent references to actual files on a file system, but allow for linking them to other LinkAhead entities and, e.g., adding custom properties.

`Properties` are individual pieces of information that have a name, description, optionally a physical unit, and can store a value of a well-defined data type. `Properties` are attached to `RecordTypes` 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 1, an example data model is shown in the right column in a UML-like diagram: There are three RecordTypes (Project, Person, and Experiment), each with 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 a diamond show references between RecordTypes, i.e., where RecordTypes are used as Properties in other RecordTypes.

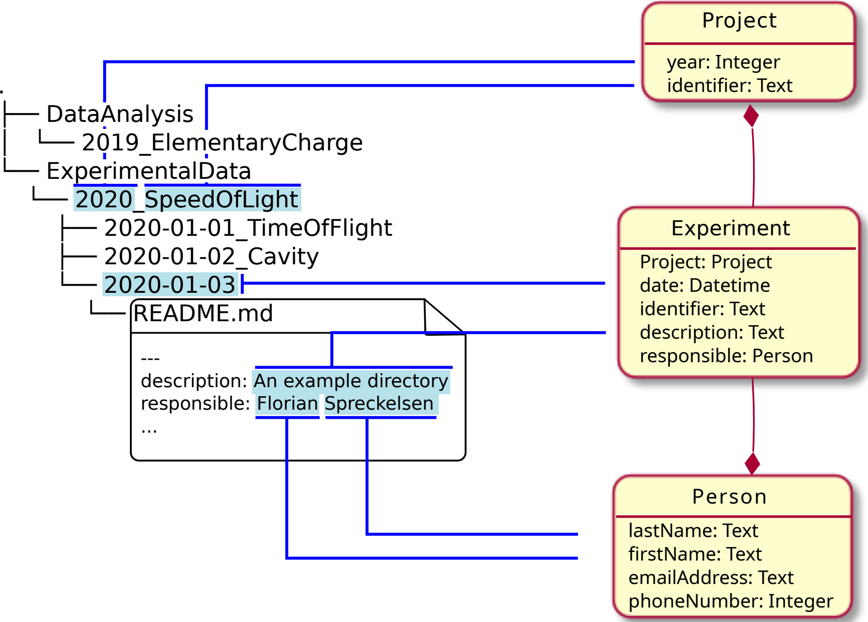

Figure 1. Mapping between file structure and data model. Blue lines indicate which pieces of information from the file structure and file contents are mapped to the respective properties in the data model.

2. Results

We solved the issues described in Section 1.1 using a modular crawler system, which is discussed extensively in this section. The main task of the crawler is to automatically synchronize information found in the file system to the semantic RDMS. The modularity of the crawler is achieved by providing a flexible configuration of synchronization and mapping rules in a human- and machine-readable YAML format. Using these configuration files (which we will refer to as CFoods), it is possible to adapt the crawler to heterogeneous use cases. These crawler definitions will be described in Section 2.2. We assumed that a semantic data model that is appropriate for the data structures has been created. As an example, we discuss the data model that is shown in Figure 1 in the right column.

In order to be able to synchronize information from the file system with the RDMS, a specification of the identity of the objects is needed. This 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 2.3. Our implementation of the procedure, the LinkAhead crawler, makes use of these concepts in order to integrate data into the RDMS. Based on the example we introduced in Section 1.2, 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. The LinkAhead Crawler

Figure 2 provides an overview over the complete data integration procedure with the LinkAhead crawler. In Step 1, the scanner uses the YAML crawler definition (Section 2.2) to match converters to a given file system tree. From the information that is matched, a list of LinkAhead Records with Properties is created. In Step 2, the crawler checks which of these Records are already contained in LinkAhead 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 2.3). In Step 3 both lists are synchronized with the LinkAhead server, i.e., all Records from the list of new Records are inserted into LinkAhead and all Records from the list of changed Records are updated. The data model (Section 1.3.1) is needed to write a valid YAML crawler definition and to create the definition of Identifiables (Section 2.3). Furthermore, it is used by the LinkAhead server directly.

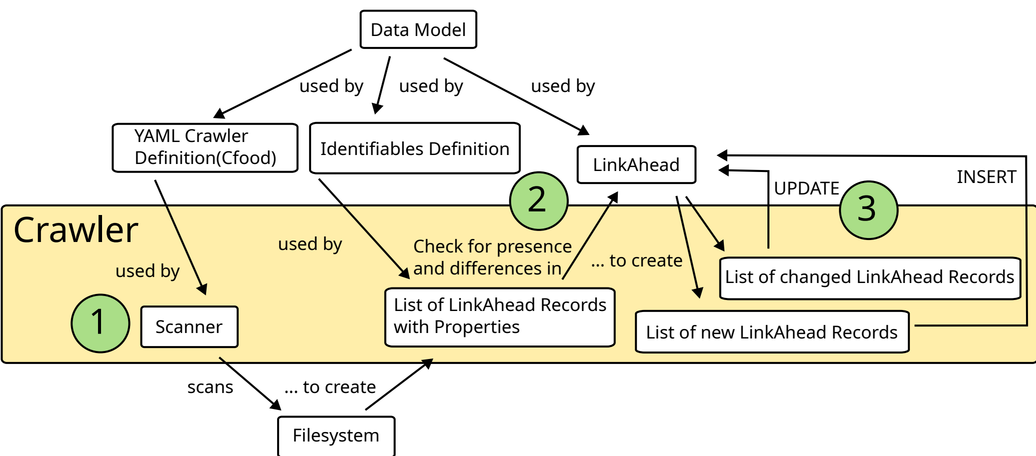

Figure 2. Overview of the complete data integration procedure. See Section 2.1.

2.2. 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 hierarchical data formats like HDF5 [24,25] files (or a mixture of both) would also fit the use case.

Figure 1 illustrates what such a mapping could look like in practice using the file structure introduced in Section 1.2:

- 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 implicitly 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.
- Each experiment contains a file called `README.md` storing text and metadata about the experiment, according to the standard described in Section 1.2. 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 the value for the property “responsible” of the `Experiment Record`.

2.2.1. YAML Definitions

Figure 3 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`) that define how these structures are interpreted and synchronized with LinkAhead. We explain this description using an example of `CFood` in this section. `CFood` is designed to make use of the example file structure shown in Section 1.2.

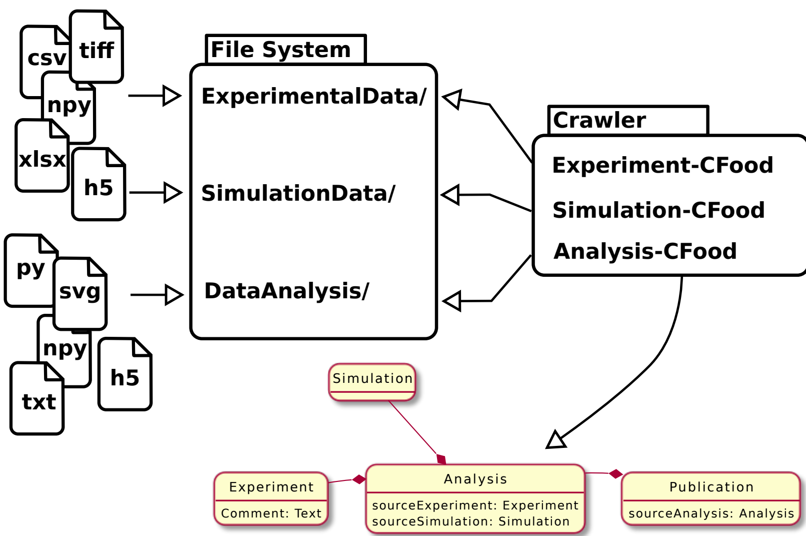

Figure 3. Illustration of the data synchronization procedure using a crawler: Data acquisition (possibly including computer simulations or data produced during data analysis) leads to a variety of files in different formats on the file system. Crawler plugins ("CFoods") are designed in a way that they understand the local structures on the file system. The file tree is traversed, 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 2.3. The figure was previously published in [26].

As we pointed out in Section 1.1, one major challenge is allowing for adapting the data integration to very different and potentially very heterogeneous use cases. Therefore, we designed a special syntax in YAML format for configuring the mapping 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 file structure the corresponding YAML file would look like this:

```
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 the 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 the `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 for 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 LinkAhead crawler. Examples include converters for interpreting tabular data (in Excel or CSV format), JSON [27,28] files, or HDF5 [24,25] files. Custom converters can be created in Python using the LinkAhead crawler Python package. There is a community repository online where community extensions are collected and maintained (see Appendix C).

2.2.2. Variables

Variables begin with a dollar sign and can be used in multiple occasions within the `CFood`, e.g., for setting properties of `Records`. These variables and their values are set during

multiple operations within the crawling procedure: Variables that are part of a matched regular expression will be available. For example, `$year` is set from the `match` entry of the `Converter Project_Dir`. Record definitions within the crawler definition will be available as variables, e.g., the `records` section of `Project_Dir` creates a variable `$Project`, which will later be used as the value of a `Property` of `Record Experiment` created in the `records` section of `Experiment_Dir`. More details about the syntax will be made available in the official documentation of the LinkAhead crawler software (see Appendix D).

2.2.3. 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 LinkAhead `Records`, where the values of the `Properties` are set to the information found. In the example shown in Figure 1, the list will contain at least two `Project Records` with names "ElementaryCharge" and "SpeedOfLight"; three `Experiment Records` with dates "2020-01-01", "2020-01-02", "2020-01-03"; and 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. Section 2.3 describes how the updates are distinguished from the insert operations. Section 2.4 describes the final process of carrying out the transactions.

2.3. Identifiables

In order to determine which of the `Records` that have been generated during the scanning procedure in Figure 1 are already present in the RDMS, the crawler needs information on how to determine identity of `Records`. For this purpose, we implemented 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 1, we can claim that each `Person` in our RDMS can be uniquely identified by providing a "firstName" and a "lastName". It is important to point out that the definition of identities for `RecordTypes` can vary highly depending on the usage scenario and environment: There might, of course, be many cases where this example definition is not sufficient, because people 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` for `RecordType Person`. We use the term "Identifiable" here, because we want to avoid confusions with the concept of "unique keys", which shares some similarities, but are also different in some important aspects. Furthermore, the RDMS LinkAhead also makes use of unique keys as part of its usage of a relational database management system in its back-end. During scanning, `Identifiables` are filled with the necessary information. In our example from Figure 1, "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 LinkAhead. 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 LinkAhead, it is retrieved. We refer to the retrieved entity as `Identified Record`. This entity will then be updated as described in Section 2.4. The terminology, which we introduce here, is summarized in Figure 4.

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, which allow for 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 what is contained in the Identifiable. This information is stored in other Properties, which are not part of the Identifiable.

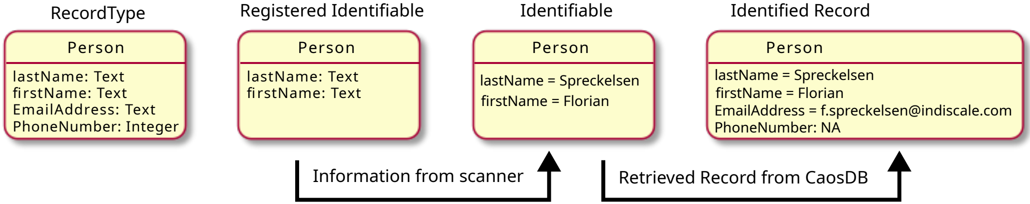

Figure 4. Terminology used in the context of defining identity for Records: For each RecordType that is used in the synchronization procedure, its identity needs to be defined in what we call the Registered Identifiable. The scanner will fill the values of the properties of the Registered Identifiable and, in that process, create an Identifiable. This Identifiable can be used by the crawler to run a query on LinkAhead. If a Record matching the properties of the Identifiable already exists in LinkAhead, it will be retrieved and used, as described in Section 2.4, to update the Record. Otherwise, a new Record will be inserted.

2.3.1. Ambigiously Defined Identifiables

Checking the existence of an Identifiable in LinkAhead is achieved 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 provided information is not sufficient to uniquely identify an entity. One example might be that there are in fact two people 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.

2.3.2. 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 using 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.

To provide an example, we discuss a hypothetical Record of RecordType Experiment that stores information about a hypothetical experiment carried out in a work group with multiple laboratories. Let us assume that there is only equipment available for one experiment per laboratory and that the maximum number of experiments is one per day. A reasonable set of minimum information for the identifiable could be the date of the experiment and the room number of the laboratory, as it uniquely identifies each experiment. Additional properties, like name of the experimenter, should not be added to the Identifiable, as it would reduce flexibility in, e.g., corrections or updates of the respective Records.

2.4. 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 2 as Step 3. As described in Section 2.3, Identifiables can be used to separate the list of Records, which were created by the scanner in Step 1, 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 RDMS.

3. Discussion

In Section 2, we presented a concept for data integration that allowed for using the file system simultaneously with an RDMS for managing heterogeneous scientific data using a modularized crawler. It was discussed that the YAML crawler definitions provided a standardized configuration that could be adapted to very different use cases.

In this section, we discuss some important design decisions, benefits of the approach, and limitations.

3.1. YAML Specifications as an Abstraction Layer for Data Integration

In Figure 1, our approach to map the 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 to 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 compromise 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 a plain data integration source code. Furthermore, the crawler specifications are machine-readable and, therefore, open the possibility for much more complex applications.

3.2. Primary Source of Information

Our update procedure synchronizes data from the file system into the RDMS unidirectionally, so we can state that the file system is the single source of truth. Another possibility would have been to implement merges between entities generated from the file system information and entities present in the RDMS. This would have allowed for editing entities simultaneously on the file system and in the RDMS, and to 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 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 considered it best practice to not edit entities generated by the crawler in the RDMS directly, but to use references to these entities instead. This could be enforced by LinkAhead by automatically setting the entities generated by the crawler as read-only for other users. LinkAhead entities are protected against accidental data loss using versioning of entities.

3.3. 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 has been criticized that this can involve a lot of work. A frequent suggestion is to instead 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 [1]. A semantic data model created by hand results in researchers understanding their data. Data organized by artificial intelligence

is very likely not represented in a way that is easily understood by the researcher and it probably 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.

One disadvantage of machine learning methods is that many of them 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 in the form of crawler definitions 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 part, manual, 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 on by the researcher.

3.4. Limitations

3.4.1. 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 the deletion of files and folders by special files (e.g., special names or contents) that trigger an automatic deletion by the crawler.

3.4.2. 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 RDMS would also be detected and propagated to the file system, leading to the creation and updating of files. While some parts of these procedure, like identifying changes in 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 present a structured approach for data integration from file systems into RDMS. Using a simplified example, we have shown how this concept is applied practically and we have published an Open Source software framework as one implementation of this concept (see Appendix A). In multiple active data management projects (some of them are mentioned in the Appendix B), we found that this mixture of a 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 the re-use of data integration modules. In these projects, we experienced that using the standardized YAML format that was presented in Section 2.2.1 highly facilitated the integration of data sets, especially in cases where data were stored in non-standardized directory layouts. We were

also able to re-use our `CFoods`, which only required minor adaptions when transferring them to a different context. To foster re-usability, we set up a community repository for `CFoods` (see Appendix C).

Our current focus for the software is on transferring the concept to multiple different use cases in order to make the crawler more robust and to identify the limitations and usability issues. Furthermore, the bi-directional synchronization that was discussed in Section 3.4.2 and an assistant for creating `CFoods` that is based on machine learning (discussed in Section 3.3) are currently under investigation as possible next features.

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 from 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. acknowledges financial support from the DZHK and DFG SFB 1002 Modulary Units in Heart Failure, and the Else Kröner-Fresenius Foundation (EKFS).

Institutional Review Board Statement: Not applicable.

Data Availability Statement: Not applicable.

Acknowledgments: The authors thank Birte Hemmelskamp-Pfeiffer for providing information on https://dataportal.leibniz-zmt.de, accessed on 2 January 2024.

Conflicts of Interest: A.S. and H.t.W. are co-founders of IndiScale GmbH, a company providing commercial services for LinkAhead. 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
PID	Persistent Digital Identifier
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 LinkAhead-Open Source project: https://gitlab.com/linkahead, accessed on 2 January 2024.
- Repository of the LinkAhead-Crawler: https://gitlab.com/linkahead/linkahead-crawler, accessed on 2 January 2024.

The installation procedures for LinkAhead and the crawler framework are provided in their respective repositories. Also a docker container is available for the instant deployment of LinkAhead.

Appendix B. Example Crawlers

There is a documented example available online at (https://gitlab.com/linkahead/crawler-extensions/documented-crawler-example, accessed on 2 January 2024) that demonstrates the application of the crawler to example data. This can also be used as a template for the development of custom crawlers. We are currently aware of one public instance of LinkAhead, which makes use of a complex crawler based on the crawler framework described in this article. It is provided by "ZMT-Leibniz Centre for Tropical Marine Research" and can be accessed online: https://dataportal.leibniz-zmt.de/, accessed on 2 January 2024.

Appendix C. Community Repository for Crawler Extensions

In order to foster the re-usability of crawler definitions, we are building a community repository for crawler extensions, which can be found at: https://gitlab.com/linkahead/crawler-extensions, accessed on 2 January 2024.

Appendix D. Software Documentation

The official documentation for LinkAhead, including an installation guide, can be found at: https://docs.indiscale.com, accessed on 2 January 2024. The documentation for the crawler framework that is presented in this article can be found at: https://docs.indiscale.com/caosdb-crawler/, accessed on 2 January 2024.

References

1. 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.; et al. The FAIR Guiding Principles for scientific data management and stewardship. *Sci. Data* **2016**, *3*, 160018. [CrossRef] [PubMed]
2. Deutsche Forschungsgemeinschaft. Guidelines for Safeguarding Good Research Practice. Code of Conduct, 2022. Available in German and in English. Available online: https://zenodo.org/records/6472827 (accessed on 10 August 2023) [CrossRef].
3. 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; Helmholtz Open Science Office: Leipzig, Germany, 2022.
4. Gray, J.; Liu, D.T.; Nieto-Santisteban, M.; Szalay, A.; DeWitt, D.J.; Heber, G. Scientific data management in the coming decade. *Acm Sigmod Rec.* **2005**, *34*, 34–41. [CrossRef]
5. Samuel, S. Integrative Data Management for Reproducibility of Microscopy Experiments. In *The Semantic Web*; Blomqvist, E., Maynard, D., Gangemi, A., Hoekstra, R., Hitzler, P., Hartig, O., Eds.; Springer International Publishing: Cham, Switzerland, 2017; pp. 246–255.
6. Vaisman, A.; Zimányi, E. *Data Warehouse Systems*; Springer: Berlin/Heidelberg, Germany, 2014. [CrossRef]
7. Barillari, C.; Ottoz, D.S.; Fuentes-Serna, J.M.; Ramakrishnan, C.; Rinn, B.; Rudolf, F. openBIS ELN-LIMS: An open-source database for academic laboratories. *Bioinformatics* **2016**, *32*, 638–640. [CrossRef] [PubMed]
8. Hewera, M.; Hänggi, D.; Gerlach, B.; Kahlert, U.D. eLabFTW as an Open Science tool to improve the quality and translation of preclinical research. *F1000Research* **2021**, *10*, 292. [CrossRef] [PubMed]
9. Suhr, M.; Lehmann, C.; Bauer, C.R.; Bender, T.; Knopp, C.; Freckmann, L.; Öst Hansen, B.; Henke, C.; Aschenbrandt, G.; Kühlborn, L.K.; et al. Menoci: Lightweight extensible web portal enhancing data management for biomedical research projects. *BMC Bioinform.* **2020**, *21*, 582. [CrossRef] [PubMed]

10. Bauch, A.; Adamczyk, I.; Buczek, P.; Elmer, F.J.; Enimanev, K.; Glyzewski, P.; Kohler, M.; Pylak, T.; Quandt, A.; Ramakrishnan, C.; others. openBIS: A flexible framework for managing and analyzing complex data in biology research. *BMC Bioinform.* **2011**, *12*, 468. [CrossRef] [PubMed]
11. Dudchenko, A.; Ringwald, F.; Czernilofsky, F.; Dietrich, S.; Knaup, P.; Ganzinger, M. Large-File Raw Data Synchronization for openBIS Research Repositories. In *Challenges of Trustable AI and Added-Value on Health*; IOS Press: Washington, DC, USA, 2022; p. 409.
12. McBride, B. The resource description framework (RDF) and its vocabulary description language RDFS. In *Handbook on Ontologies*; Springer: Berlin/Heidelberg, Germany, 2004; pp. 51–65.
13. *OWL 2 Web Ontology Language Document Overview*, 2nd ed.; World Wide Web Consortium: Boston, MA, USA, 2012; p. 7.
14. Pérez, J.; Arenas, M.; Gutierrez, C. Semantics and Complexity of SPARQL. *ACM Trans. Database Syst.* **2009**, *34*, 1–45. [CrossRef]
15. Bizer, C.; Heath, T.; Ayers, D.; Raimond, Y. Interlinking Open Data on the Web. In Proceedings of the 4th European Semantic Web Conference, Innsbruck, Austria, 3–7 June 2007; p. 2.
16. Bizer, C.; Heath, T.; Berners-Lee, T. Linked Data—The Story So Far. In *Semantic Services, Interoperability and Web Applications: Emerging Concepts*; IGI Global: Hershey, PA, USA, 2011; p. 26.
17. De Smedt, K.; Koureas, D.; Wittenburg, P. FAIR Digital Objects for Science: From Data Pieces to Actionable Knowledge Units. *Publications* **2020**, *8*, 21. [CrossRef]
18. Vassiliadis, P. A Survey of Extract–Transform–Load Technology. *Int. J. Data Warehous. Min. (IJDWM)* **2009**, *5*, 75. [CrossRef]
19. 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. [CrossRef]
20. Hornung, D.; Spreckelsen, F.; Weiß, T. Agile Research Data Management with Open Source: CaosDB. 2023. Available online: https://www.inggrid.org/article/id/3866/ (accessed on 2 January 2024).
21. 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. [CrossRef]
22. 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.; et al. The brain imaging data structure, a format for organizing and describing outputs of neuroimaging experiments. *Sci. Data* **2016**, *3*, 160044. [CrossRef] [PubMed]
23. Mildenberger, P.; Eichelberg, M.; Martin, E. Introduction to the DICOM standard. *Eur. Radiol.* **2002**, *12*, 920–927. [CrossRef] [PubMed]
24. Koranne, S., Hierarchical Data Format 5 : HDF5. In *Handbook of Open Source Tools*; Springer: Boston, MA, USA, 2011; pp. 191–200. [CrossRef]
25. Folk, M.; Heber, G.; Koziol, Q.; Pourmal, E.; Robinson, D. An overview of the HDF5 technology suite and its applications. In Proceedings of the EDBT/ICDT 2011 Workshop on Array Databases, Uppsala, Sweden, 25 March 2011; pp. 36–47.
26. Schlemmer, A. *Mapping Data Files to Semantic Data Models Using the CaosDB Crawler*; Zenodo: Geneva, Switzerland, 2021. [CrossRef]
27. Pezoa, F.; Reutter, J.L.; Suarez, F.; Ugarte, M.; Vrgoč, D. Foundations of JSON schema. In Proceedings of the 25th International Conference on World Wide Web, Montreal, QC, Canada, 11–15 April 2016; pp. 263–273.
28. Bray, T. *The Javascript Object Notation (Json) Data Interchange Format*; Technical Report; 2014. Available online: https://datatracker.ietf.org/doc/rfc7159/ (accessed on 10 August 2023).

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.