University of Passau
Faculty of Computer Science and Mathematics

Bachelor thesis

Implementation of a distributed environment of medical datasources

David Goeth

Bachelor thesis
Chair of Distributed Information Systems
Faculty of Computer Science and Mathematics
University of Passau

Examiner: Prof. Dr. Harald Kosch
Supervisor: Armelle N. Ndjafa

March 29, 2018

Abstract

Health care faces the problem of dealing with a vast amount of heterogenous data that isn't easily accessible. A problem producing lots of costs. It would be helpful to have a system facing this issue. Classical data integration approaches need a global schema before queries on the integrated data can be stated. Data integration approaches work, but are cumbersome. Dataspace is a new abstraction of data management, that doesn't enforce any schema, and doesn't enforce full data integration in query results. Thus, a dataspace allows e.g. that semantical equal data is not presented equal. This is also called the acceptance of *data co-existence*. A dataspace also needs only low upfront work for datasources, thus being suitable for highly heterogeneous environments like the healthcare sector. In this thesis we present MeDSpace, a distributed system that allows to state keyword search queries over heterogeneous datasources. The presented system is a test environment for medical datasources and can be used as a starting point for creating a dataspace over medical datasources. The system uses medical data, but could be used for any kind of multimedia data that should be searchable by keywords.

Abstract (german translation)

Der Gesundheitsbereich ist mit dem Problem konfrontiert, dass eine gewaltige Menge von heterogenen Daten existiert, die nicht einfach so abrufbar sind. Es wäre daher hilfreich ein System zu haben, das dieses Problem schmälert. Klassische Lösungen zur Informationsintegration brauchen ein globales Schema, bevor Anfragen an die integrierten Daten gestellt werden können. Auch wenn solche Ansätze funktionieren, sind sie doch sehr schwerfällig und können nicht dynamisch an die ständigen Veränderungen im Gesundheitsbereich angepasst werden, ohne erheblichen Verwaltungsaufwand. Dataspace ist eine neue Abstraktion für Datenmanagement, das kein globales Schema zwingend benötigt und die Koexistenz von Daten im Integrationssystem erlaubt. Zusätzlich wird ein verhältnismäßig geringer Arbeitsaufwand benötigt, um Basisdienste in einem Dataspace anbieten zu können. Das macht ein Dataspace attraktiv für besonders heterogene Umgebungen wie für den Gesundheitsbereich.

In dieser Arbeit präsentieren wir MeDSpace, ein verteiltes System, das es ermöglicht, Stichwortsuchanfragen an eine Menge von heterogenen Datenquellen zu stellen. Das System ist eine Testumgebung für medizinische Datenquellen und kann als Startpunkt verwendet werden, um ein Dataspace über medizinische Datenquellen zu entwickeln. Auch wenn das System medizinische Datenquellen verwendet, kann es trotzdem für jegliche Art von Multimediadaten verwendet werden, die über eine Stichwortsuche verfügbar sein sollen.

I want to thank...

Armelle N. Ndjafa for her support and supervising,

Florian Duschl and Gerina Goeth for their effort to proofread,

and Sarah Goeth for giving motivation.

Contents

List of Tables

List of Figures

1 Introduction

In the fields of medicine every year a vast amount of (digital) data is produced, that usually is very complex [6, p. 1]. The data in the healthcare sector are combinations of classical text files, images, audio, and videos that is used in combination. Thus the data meets the definition of multimedia data, which is exactly a conjunction of multiple kinds of data that is used to present modal information [7, p. 2].

Health data is also highly heterogeneous (e.g. different file formats) and and often requires lots of tools to work with.[8, p. 1]. In summary, searching data and working with it isn't a straightforward process, and indeed causes many costs.

As a result, the ability to easily access the data would be beneficial for both, research and healthcare institutions [8, p. 2]:

- Lower Costs
- Detecting diseases at early stages
- Simplified collaboration
- Healthcare fraud detection

The above mentioned benefits were originally stated for Big Data Analysis and Data Mining. Both research fields require as a first step to integrate the data of different datasources into one data integration system. Creating a data integration system for the healthcare sector would not only be reasonable, but offers lots of necessary benefits to the whole healthcare system.

But there is another option, the so-called dataspace concept, that doesn't semantically integrate data before it is able to provide its services, and follows a data co-existence approach. Data co-existence means that data from a query result isn't fully integrated. So it is allowed e.g. that semantical equal data is presented differently. Duplicates in a query result are thus acceptable. Data co-existence facilitates work with heterogeneous data and many schemas. As heterogeneity is a first-class citizen in a dataspace, it is also useful for integrating multimedia data. These are attractive properties and reason in this work for to follow the dataspace concept . The dataspace concept was introduced in 2005 as a vision [9]. Yet, until today no fully fledged dataspace implementation was presented.

In this thesis we present a distributed environment to state keyword search queries on multiple heterogeneous multimedia medical datasources. We called this system MeDSpace which is an abbreviation for *Medical Dataspace.* The system could be used as a base for implementing a fully fledged dataspace or for any system, that needs keyword search functionality on multiple heterogeneous (distributed) datasources. Despite the research of dataspaces is very active, they often cover only text data ignoring the properties of multimedia data [10]. This is another reason, why we explicitly chose multimedia data.

There should be noted, that although we use medical data for test data of the system to be presented, the system itself isn't restricted to any kind of particular data.

The content of the thesis is structured as follows: In chapter 2 we will present foundation knowledge of data integration and in chapter 3 we discuss basics of dataspaces. Then we will cover state-of-the-art projects in chapter 4. In chapter 5 the structure and the provided services of MeDSpace is presented and in chapter 6 the implementation of our system is described in detail. At the end, in chapter 6 we will give some outlook, suggestions, and possible improvements to the system.

2 Foundation in Data Integration

Data integration constitutes the issue of combining data residing at different sources and providing the user with a unified view of these data [11].

We have now a definition of the concept data integration, but for what do we need it? The goal of data integration is almost always to simplify the access to a range of existing information systems through a central, integrated component with a unified interface for users and applications. Therefore, integrated information systems provide a unified view on the datasources. Existing datasources can be diverse: Classical relational database systems, files, data accessed by web services or HTML formulas, data generating applications or even other integrated information systems [1, p. 3-4].

In general, there are two different types of data integration: The *materialized* integration and the *virtual* integration. The difference between these two approaches reads as follows:

- in case of *materialized* integration the data to be integrated is stored into the integrated system itself. The data in the datasources remain, but for querying is used the materialized view. A classic example of a materialized data integration system are data warehouses.
- in case of *virtual* integration, the data are transported from the data source to the integrated system while the query processing. That temporary data is then again discarded. So, integration isn't done once but on each query.

Of course, an integrated information system can use both principles. A system of this kind is called *hybrid*. Both types have in common that a query is processed on a global schema. For the virtual integrated system, the data only exists virtually, thus relations between data sources and the global schema have to be specified. Then, on query time the query has to be split into query schedules. The schedules are responsible for extracting the needed information from the different data sources and subsequently merge and transform the data [1, p. 86-88].

Today, data is classified into three diverse categories:

Structured data	Have a predefined structure through a schema.
Unstructured data	Contains no predefined structure.
Semi-structured data	This kind of data can also have a schema, but can deviate from the schema, too.

Table 2.1: Types of data [1, p. 17]

A common example for structured data are relational data. Unstructured data can be seen as the opposite of structured data. Typical unstructured data is natural language text. Semi-structured data are in the middle between the other two, i.e. semi-structured data have structured data, but aren't forced to use a schema. An example of semi-structured data is an XML file without an accompanying XML schema.

A significant property to consider when looking at data is heterogeneity. What heterogeneity is and how it affects data integration will be explained in the next section. But before we explain it, we should introduce Some important terms and concept that we need to understand data integration and heterogeneity:

Data model: In data modeling theory a data model is an abstract model that describes how data is represented and used. It consists of a set of data structures and conceptual tools that is used to describe the structure of a database and operations that can be performed on the data. A data model consists of a data model theory, a formal description of the data model, and a data model instance, which is a practical data model designed for a particular application. There exist three types of data models [12, p.10-14]:

- **Conceptual data model**: Describes data independent from any implementation details. It is used to describe the semantic of a domain. Examples for conceptual data models are the entity-relationship (E-R) model and ontologies.
- **Logical data model**: Also called representational or implementation data model. Describes data in terms of data structures, like relational tables and columns or XML tags. Examples of logical data models are the relational data model, the object-based data model, semi-structured data models like XML or a graph-based data model like RDF.
- **Physical data model**: Describes data in terms of collection of files, indices and other physical storage structures. It defines how the data is stored on disk and the available access methods.

Data source: A data source addresses an arbitrary data storage, whose data should be integrated in an information system [1, p. 7].

Integrated or integrating information system: An integrated information system is an application, which facilitates the access to different data sources [1, p. 7].

Meta data: Data that describe other data. The distinction between data and meta data in a system depends on the particular application. Meta data has to be stored, browsed and integrated as well as 'normal' data [1, p. 8].

Schema: The word comes from greek *σχήμα* (skhēma) meaning *shape* or *plan*. In computer science this word has different meanings. In the context of database theory the word is used for a representation of the structure (syntax), semantics, and constraints on the use of a database (or its portion) in a particular data model. [13, p. 235]. Analogous, an XML schema defines the structure, content and semantics of XML documents[14]. To generalize it, we can define a schema in the context of data integration as the definition of the structure, content and semantics of a data source.

2.1 Heterogeneity

Information systems providing not the same methods, models and structures for accessing their data are called *heterogeneous* [1, p. 58]. It commonly occurs that distributed systems (and therefore are maintained independently) tend to be heterogeneous, since they are managed autonomously. In other words, distribution often leads to heterogeneity, but not necessarily. De facto, it is to observe, that data sources tend to be the more heterogeneous the more autonomous they are. Two independent systems will in practice always be heterogeneous, even if they contain the same kind of data. Heterogeneity arises from different requirements, different developers and different temporal developments. It even occurs if initially identical software systems are used, but over time they were adapted to the specific enterprise's needs. This process is called *customizing* [1, p. 59].

Heterogeneity is the main issue of data integration [1, p. 59]. As a consequence, integrated systems often restrict autonomy of the data sources making them more homogeneous. A common example would be the use of industry-specific standards like common exchange formats, interfaces or communication protocols.

Heterogeneity exists between the data sources, as well between the data sources and the integrated system [1, p. 60]. But often only the latter case is relevant, as the data sources often don't communicate among themselves. To bridge heterogeneity, it obviously is necessary to translate queries and to implement missing functionality in the integrated system. An example of heterogeneity between a data source and the integrated system is, if the data source provides SQL-access, but the integrated system uses SPARQL for querying.

Heterogeneity is an issue we have to overcome in the thesis' project. It is necessary to have a thorough understanding of the nature of heterogeneity in order to overcome it correctly. Thus, we will discuss heterogeneity more in depth in the next sections. Heterogeneity can be split into different types: Table 2.2 shows an overview of the existing kinds of heterogeneity [1, p. 60/61].

Technical heterogeneity	Includes all technical problems to realize the access of the data from the datasources.
Syntactic heterogeneity	Includes problems in the presentation of data.
Data model heterogeneity	means problems in the presentation of data of the used data models.
Structural heterogeneity	Includes differences in the structural representation of information. This heterogeneity is solved if semantic identical concepts are also structural equally modeled.
Schematic heterogeneity	Important special case of the structural heterogeneity, whereby there are differences in the used data model.
Semantic heterogeneity	Includes problems regarding the meaning of used terms and concepts.

Table 2.2: Kinds of heterogeneity [1, p. 60/61]

2.1.1 Technical heterogeneity

This kind of heterogeneity defines differences between information systems to access their data. There are different technical layers on which heterogeneity can exist [1, p. 62]:

- **Request function:** query language, parameterized functions, canned queries (forms)
- **Query language:** SQL, XQuery, SPARQL, full text search
- **Exchange format:** binary data, XML, HTML, tabular
- **communication protocol:** HTTP, JDBC, SOAP

Technical heterogeneity can be further subdivided in [1, p. 64]:

- **Access heterogeneity:** Authentication and authorization differences between information systems
- **Interface heterogeneity:** Differences in the technical realization of data source access. A special case here is heterogeneity in the query mechanism, e.g. the used query language.

Technical heterogeneity is overcome if the integrated system is able to send a query to a data source and that data source principally understands the query and produces a set of data as a result [1, p. 61].

2.1.2 Syntactic heterogeneity

Syntactic heterogeneity describes differences in the presentation of the same facts[1, p. 64]. For example, different number formats (little endian and big endian), different character encodings (Unicode, ASCII) or different separators in text files (tab delimited and comma separated values (CSV)). Syntactic heterogeneity therefore can be reduced to technical differences in the presentation of information, whereas the *synonym problem,* which deals with the representation of equal concepts through different names, is an issue of *semantic heterogeneity.*

This kind of heterogeneity is overcome if all data meaning the same are presented equally [1, p. 61].

2.1.3 Data model heterogeneity

Data model heterogeneity arises if the integrated system and a data source manage their data in different data models [1, p. 65]. Data model usually means the *conceptual* or the *logical* data model (or both), as the physical data model is only used for internal processing inside the data source and never propagated outwards.

Important to note is the fact, that data model heterogeneity is independent from *semantic* differences. Two systems can use two different data models both describing the same domain, e.g. describing data in the object oriented or relational data model. Nevertheless, it is to observe that differences in the data model often induces semantic heterogeneity.

Data model heterogeneity is solved if the data sources and the integrated system use the same data model [1, p. 61].

2.1.4 Structural heterogeneity

Using the same data model doesn't mean semantic equal data has to be described equally. We speak of structural heterogeneity if two schemas are different, even though they represent the same extract of the real world (i.d. the intension of their schema elements is equal) [1, p. 67].

Structural heterogeneity can have many reasons like different preferences of developers, different requirements, using different data models, technical restrictions, etc. . This arises from the *design autonomity* of data sources. A Data source has design autonomity if it can freely decide in which way it provides its data, including the data format, the data model, the schema, syntactic represenation of data, the use of concept systems, and the units of values [1, p.55].

This heterogeneity is solved if semantic identical concepts are also structural equally modeled [1, p. 61].

2.1.5 Schematic heterogeneity

A special case of structural heterogeneity is schematic heterogeneity [1, p. 67]. Schematic heterogeneity is present if different elements of the data model are used to model the same thing. In the relational model it is possible to model information as relations, as attributes or as values, for instance.

Schematic conflicts are particularly difficult to resolve, as it isn't usually possible to overcome them with the query language [1, p. 68]. For example, it is mandatory to explicitly state attributes and relations in relational languages. If schematic heterogeneous data sources should be integrated, one would define a view. But the query has to be altered if something changes in the elements.

2.1.6 Semantic heterogeneity

Individual values have no implicit meaning in an information system. Looking at the number '20' on a table doesn't allow conclusions concerning its meaning. In fact, it could be anything. Only by *interpreting* a datum it gets a meaning, and for interpretation one needs knowledge about the concrete use case and world knowledge. The interpretation of data is also called *semantic*. Therefore, to interpret data in an information system, further information has to be consulted [1, p. 73]:

- The name of the schema element containing the data
- The position of the schema element within the schema
- Knowledge about the domain the schema is developed for.
- Other data values stored in the schema element

Comprehending all this information, we speak of the data's *context* [1, p. 74]. Data only obtain its meaning by taking its context into account. But it should be considered that some parts of the context are given in machine-readable form (like the schema), and some are not (e.g. domain specific knowledge). Also absolutely identical schemas with different contexts can have different semantic meanings. Owing to circumstances, an integrated system has to make implicit context knowledge explicit, e.g. by introducing new schema elements.

Semantic conflicts relate to the interpretation of *names* and *symbols*, respectively [1, p. 74]. Semantic conflicts occur in the interaction of names and concepts in different systems. In this context, a *concept* means the intension of a name. The set of real world objects of a name is called its *extension*. The most common semantic conflicts are synonyms and homonyms [1, p. 75]. Two names are *synonyms* if they have the same intension, but are syntactic different. An example of synonyms are the words *disease* and *illness*. They mean the same but are different words.

Whereas two names are *homonyms* if they are syntactically identical, but their intensions are different. Example: Two tables contain a column called *date*. The first table stores dates about the beginning of a treatment whereas the other table stores dates of births.

More difficult to treat are names, though, whose intensions neither are identical nor completely different, but conclude themselves or are overlapping. Example: We have two names *patient* and *person*. Not every person is automatically a patient and also a patient needn't to be also a person (e.g. an animal could be a patient in the veterinary medicine). But in certain contexts, *patient* and *person* can be seen to be equal.

In principle, it is difficult to detect semantic conflicts and resolve them clearly [1, p. 76]. It has to be considered that often for schema analysis only the schema is available and some example data, sometimes maybe also some domain knowledge and documentation of the data sources, but that isn't always the case, particularly on autonomous systems. Especially difficult are cases with less clear defined concepts or with options for multiple interpretations.

A complete resolving of semantic conflicts isn't always possible, or only achievable with big effort, or not necessary. This is also called *remaining blur* [1, p. 76].

Semantic heterogeneity is solved if the integrated system and the data source really mean the same by the used names for schema elements. Equal names have consequently equal meaning [1, p. 61].

2.2 Distribution

An issue of data integration is the distribution of data to be integrated [1, p. 51]. We speak thereby of data which lies on different systems. Important to know, we proceed the assumption that the acccess to the data is ensured, i.e. the systems are connected.

Data distribution isn't plainly an annoyance, as it is often an intended design decision[1, p. 54]. The distribution is handy for load balancing, reliability and protection against data loss. Thereby the distribution is controlled on a central point. The consistency of data is assured using sophisticated mechanisms like the 2-phase-commit protocol[1, p. 54]. Contrary, in a typical data integration project distribution of data has historically evolved or is due to organizations and thus is uncontrolled redundant.

Distribution can be structured into logical and physical distribution [1, p. 51]. Data is *physical* distributed if it lies onto physical different systems which geographically also can be located on different places. In turn, data is *logical* distributed, if there exists multiple possible locations for storing the same data.

2.2.1 Physical distribution

In order to integrate physical distributed data several issues have to be solved [1, p. 51]: the first step is to detect the physical storage location of the data. Therefore, computer, server, port, and network have to be identifiable and locatable. For identification, applications used in the internet use Uniform Resource Locators (URLs) to identify remote computers and services not knowing their exact location.

The second problem resulting from physical distribution is data stored in different schemas [1, p. 52]. However, common query languages don't provide the possibility to query data residing in different schemas. There are two different solutions:

- Separating a query into several schema specific queries and consolidating the results in the integrated system.
- Developing a language able to handle several schemas in a posed query.

Physical distribution also changes the requirements for the query optimizer [1, p. 52]: In a central database the optimizer's main function is minimizing the accesses to the local storage, whereas for a distributed database the accesses over the network should be held as small as possible. This is important as remote accesses need considerably more time than accesses to the local storage.

2.2.2 Logical distribution

If identical data is located on different locations inside the same system, we speak of a logical distribution [1, p. 52]. The key property of logical distribution is therefore the *overlapping of the intension* of different data storage locations. Thereby, it is insignificant where these systems physically really are. Two database instances on one computer induce already distribution issues.

In this context redundany plays a major role in a system [1, p. 53]. A distributed system is called redundant if semantically identical data can be located at different places. In order to get a consistent view of a redundant system, redundancy has to be strictly controlled (e.g. using triggers or replication mechanisms). At anytime it has to be assured that the same data is at all different places. But data integration typically has to deal with *uncontrolled redundancy* for each data source is maintained independently. This fact leads to several problems [1, p. 53]:

- **Localization**: The integrated system has to provide meta data allowing to *locate* data. E.g. this can be done by a catalog of all schemas and their descriptions or a global schema with mappings to source schemas.
- **Duplicates**: If it is possible that data exists on multiple locations, there will exist duplicates, i.e. objects stored on both locations. These objects have to be recognized by the integrated system.

- **Inconsistencies**: Redundant data can contain inconsistencies that have to be resolved for a homogeneous presentation.

2.3 Architectures

In order to design a system for a data integration process, it is necessary to understand the basic architectures that we found for data integration, so that a suitable reference architecture can be chosen. For this, we will different architectures in the following.

Integrated systems evolved from distributed databases which in turn are based on the classical *monolithic database*. This system is subdivided into three layers as shown in figure 2.1 and is based on the ANSI/X3/SPARC three-layer architecture [15], [1, p. 84/85].

The *internal (or physical) view* is responsible for the storage of the data on the respective database. One layer upwards comes the *conceptual (or logical) view* modeling the data on a conceptual way. The conceptual schema defines which data model is used, which data is stored in the DBMS, and the relations among the data. Splitting the data into the internal and conceptual view makes the data independent from the physical storage medium. The top layer is called *external (or export) view* and is responsible with modeling of data as well as the conceptual view. However, it isn't modeled the whole application domain. The external view only specifies which part of the conceptual schema is provided to the respective application. An export schema is initially a subset of the conceptual schema, but can be transformed and aggregated. In the external view are defined access restrictions, too. Splitting up conceptual and external view ensures *logical data independence*.

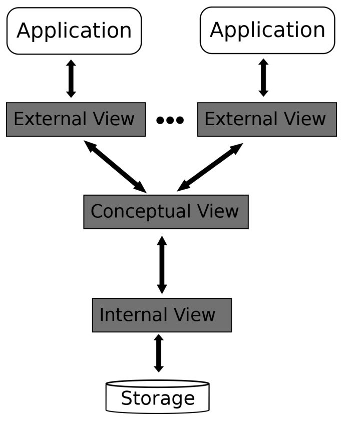


Figure 2.1: ANSI/X3/SPARC three-layer architecture [1, p. 85]

2.3.1 Distributed Systems

From the monolothic database evolved the distributed database architecture, as shown in figure 2.2.

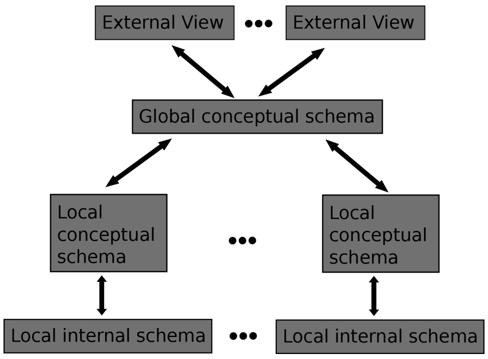


Figure 2.2: Architecture of a distributed database [1, p. 92]

The idea with distributed systems is to distribute data onto several systems (physical and logical), but it should be possible to query all data at once. In order to achieve this, the architecture is subdivided into *four layers*. Each datasource has a local internal and local conceptual schema. The latter only mirrors the data managed by the local database. Dependent on the used distribution strategy, the local conceptual schema is equal to the global conceptual schema or an extract from it. Common distribution strategies are vertical and horizontal partitioning [1, p. 92]:

- **horizontal partitioning:** Data of big tables is distributed per tuples on different computers. A union operation consolidates these parts again.
- **vertical partitioning:** Data of big tables is distributed per attributes on different computers. Each partition contains additionally a shared key attribute. This makes it possible to consolidate the partitions with a join operation.

On top of the local conceptual schemes stands a global conceptual schema. This schema models the whole application domain and is the central point of reference for the external schemes playing the same role as in the three-layers-architecture.

Distributed databases are close coupled. They are strictly controlled while conception and operation and thus the main problems of heterogeneity (like structural and semantic heterogeneity) don't occur [1, p. 93]. Thus, this architecture is good for environments, there the datasources to be distributed are less heterogeneous. For highly heterogeneous systems (like medical datasources), though, this architecture is unsuitable.

2.3.2 Multidatabase Systems

For to enable heterogeneous databases to connect, the Multidatabase System (MBS) architecture has been evolved (shown in figure 2.3).

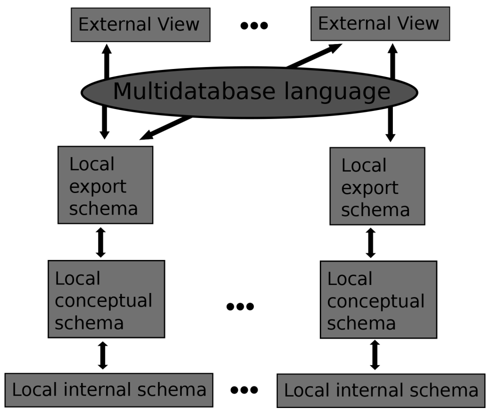


Figure 2.3: Architecture of a multidatabase system [1, p. 94]

MBSs are collections of autonomous databases being loosely linked [1, p. 93]. Each database grants external applications access to its data. The access is done using a database language which allows to query several databases in one query. A language of that kind is called multidatabase language. To obtain the autonomy of the involved databases, a MBS has no global conceptual schema. Instead, each local database keeps an export schema defining which part of the local conceptual schema is provided to external applications. It is assumed that no data model heterogeneity is contained in a MBS, i.e. all databases use the same data model, or either the multidatabase language or the local datasource provide a translation to the global data model. Now, each application can create its own external schema, which integrates one or more data sources. So, it's the task of the application doing the integration task. A MBS provides only a suitable language for querying [1, p. 94]. It can be noted that a MBS is only suitable for datasources that use the same data model. If this is not the case, another system has to be used. Furthermore should be considered: the query language has to support all languages of the data sources. In order to add a datasource with a not supported query language the MBS query language has to be altered.

2.3.3 Federated Systems

In contrast to MBSs, federated database management systems (FDBMS) have a global conceptual schema as seen in figure 2.4. The schema is also called the *federated schema.*

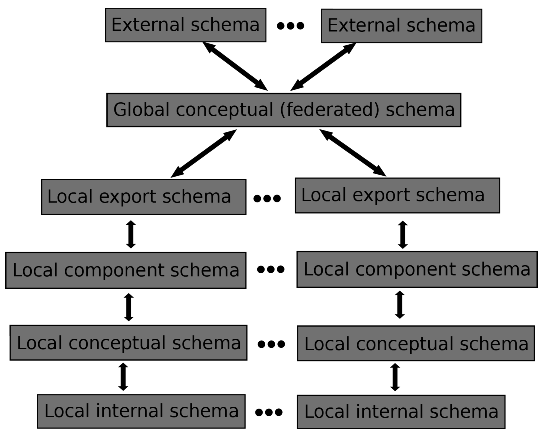


Figure 2.4: Architecture of a federated database management systems (FDBMS) [1, p. 95]

The federated schema is the central point of reference for all external schemes and there applications [1, p. 94]. But in contrast to distributed databases the global schema results after the local schemes with the goal to provide an integrated view of existing and heterogeneous data sets. Data sources keep a high degree of autonomy. The used data model in the global scheme is known as the *canonical* data model.

All in all, a FDBMS consists of five different layers [1, p. 95]:

- **Local conceptual schema**: Defines the data of a data source. Is comparable with the conceptual schema of the three-layer architecture.
- **Local component schema**: Defines the data of the local conceptual schema in the canonical data model.
- **Local export schema**: Does the same as the external view in the three-layer architecture, it defines which elements of the local component schema are visible/query-able from the outside.
- **Global conceptual/federated schema**: Defines an integrated view over all data sources and is defined in the canonical data model. The federated schema consists of multiple export schemas [13, p. 200].
- **External schema**: Defines a subset of the global schema that should be visible to an application.

Remarkable is, a federal system can have several federated schemas that integrate differently the export schemas. This allows data with different semantic meaning within the federated system when combining query results of different federated schemas. But query results of a specific federated schema are always semantically integrated, so, there is no data co-existence between them. As we will see later, this is a crucial difference in respect to dataspaces.

2.3.4 Mediator-based Systems

Mediator-based information systems are a generalization of the previous architectures (see figure 2.5) [1, p. 97]: They know only two separate components, namely Wrappers and Mediators.

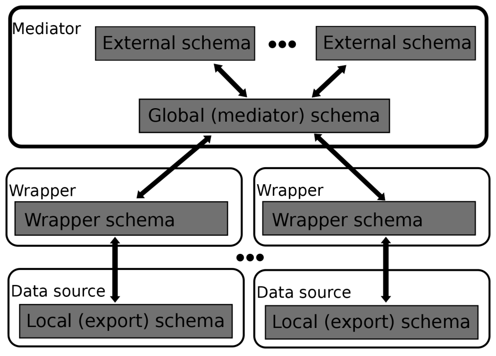


Figure 2.5: Architecture of a mediator-based information system [1, p. 97]

Wrappers are software components responsible for the access to a solely data source. A Wrapper has to break down technical, data model, schematic and interface (technical) heterogeneity. It realizes the communication between the mediators and datasources, and includes following tasks:

- overcoming interface heterogeneity (e.g. SQL to html formulars)
- overcoming language heterogeneity. This includes the handling of restricted data sources and different query languages.
- providing data model transparency through translating the data into the canonical data model.
- resolving schema heterogeneity by using a suitable mapping between the source schema and the global schema.
- supporting the global query optimizing by providing information about the query capabilities of the data source and expected costs.

The thesis' system project uses wrappers. Thus we go more in depth and look at the design, architecture and implementation goals of an architecture using wrappers. The following statements are results of from [16]:

Low start-up costs: Writing a wrapper should be done very quickly, very simple wrappers even within a few hours. The authoring of a wrapper should also be as simple as possible for writing the wrapper with little or no knowledge about the internal structure of the integrated system.

Easy evolving: Evolving should be done very simple due to two reasons: a wrapper should be implemented very fast to show feasibility. More sophisticated features a datasource can provide should be added later. Additionally the data source can change itself over time and the wrapper should be easily adaptable.

The second component of a mediator-based ssystem is the mediator software component which uses knowledge of certain data to *create and provide* information for other applications. Mediators communicate with one or more wrappers and deliver a specific value, normally structural and semantic data integration.

The following list shows an extract of the services a mediator might provide (from, [17, p. 5-6]):

- Selection of likely relevant data, e.g. used for scoring query results

- Invocation of wrappers to deal with legacy sources
- Resolution of domain term terminology and ontology differences
- Sending data and meta-data to the customer application
- Imposition of security filters to guard private data
- Handling data duplication, e.g. by deleting it
- Integration of data from several data sources

In a mediator-based architecture data sources don't know of the existence of the integrated system: only the wrapper directly speaks with the data source. As a result autonomy is preserved for all data sources [1, p. 97].

2.4 Ontologies

In computer science an ontology defines the concepts, relationships, and other distinctions being relevant for modeling a domain [18]. It operates on the semantic level and thus is independent from lower data models (logical and physical data models). Since ontologies are independent from lower level data models, they are suitable for integrating heterogeneous data sources. Data integration using ontologies is also called *ontology-based* integration.

To specify an ontology description logics are used [1, p. 267]. With description logics it is possible to define classes of a domain and the relations between these classes. An advantage is that by using description logics a domain can be specified much more precisely than using the relational data model.

Ontologies can generate new knowledge from existing data by using *logical inference*: The ontology defines basic rules in a formal model, and by applying these rules on a given data set the ontology might infer knowledge from it that wasn't explicitly stated before.

3 Dataspaces

The concept of dataspaces was the first time presented in 2005 and describes a new abstraction of data management [9, p. 27]. The motivation for a new type of data management arises from the issue that rarely the data to be managed is solely stored in a convenient relational database, nor is stored in a single data model. For a long time relational databases were the dominating choice for storing data, but this is changing, as the coming popularity of NoSQL databases indicates [19]. Also for multimedia content, relational databases aren't always an ideal choice since they were conceptualized for structured data, whereas multimedia data can contain all types of data. Also distribution is a problem, that is difficult to handle with relational databases, since they were also designed for monolithic use-cases. We can record that applications have to handle more and more different types of data sources and thus have to handle heterogeneous and distributed data, too.

That issue is ubiquitous and arises in enterprises, government agencies and even on everybody's PC desktop. As example par excellence serves the healthcare sector dealing with highly heterogeneous and complex data, and struggling with a continuously growing amount of data.

As a response to this problem, the authors of [9] postulate the concept of dataspaces and corresponding to this the development of Dataspace Support Platforms (DSSPs). Shortly said, the latter provides an environment of cooperating services and guaranties that enables software developers to concentrate on their specific application problem rather than taking care of returning issues in consistency and efficiency of huge, linked but heterogeneous data sets. The remarkable properties of a dataspace system are defined as follows [9, p. 28]:

- A DSSP must deal with data and applications in a variety of file formats that are accessible through many systems with different interfaces. A DSSP has to support all kinds of data in the dataspace and isn't allowed to omit any (e.g. as DBMSs do).
- An integrated possibility for searching, querying, updating and administration is provided by the DSSP, but the data often is also accessible and modifiable through native interfaces. Therefore DSSPs haven't full control over their data.
- The DSSP can offer several query services, but isn't forced to return the most accurate result. Indeed, the answers can be approximated resp. *best-effort*. If some data sources e.g. are unavailable for some reason, the dataspace might use the current available data sources and creates the best possible result.
- Whenever it is required, the DSSP has to provide tools to do tighter data integration.

Current data integration systems are essentially a natural extension of traditional databases, where queries are specified in a structured form and data is modeled in one of the traditional data models like relational or XML. The system for data integration also has exact information about how the data in the sources map to the schema used for integration. So, for data integration there is a need for large upfront effort in creating the mediated schema and the schema mappings.

Many of the services a dataspace provides, data integration and exchange systems provide, too. The main difference between these systems is that data integration systems need a semantic integration process before they can provide any services on the data. Additionally, data integration systems always have a global schema. But a dataspace is not a kind of a classic data integration approach. Initially, a dataspace doesn't semantically integrate the data. It allows data to *coexist*. The idea is to provide base functionality over all data sources, regardless of their specific integration constraints. For instance, a DSSP is able to provide a keyword search similar to a desktop file search. If more sophisticated operations are required, such as relational queries, data mining or monitoring of specific data sources, additional effort can be done to integrate the sources tighter. This incremental process is also called as *pay-as-you-go* data integration.

A dataspace should contain all information being relevant for a specific organization/task, regardless of their file format or storage location, and it should model a collection of relationships between the data sources. Therefore, the dataspace is defined as a set of participants and relationships [9, p. 29]. Participants of a dataspace are individual data sources. Some participants support expressive query languages for querying, while others only provide limited access interfaces. Participants can reach from structured, semi-structured right up to unstructured data sources. Some sources provide traditional updates, some are only be appendable, while others are immutable. Further, dataspaces can be nested within each other, which means that a dataspace should be able to be a part of another dataspace. Thus, a dataspace has to provide methods and rules for accessing its sub dataspaces.

Not every participant will provide all necessary interfaces for being able to support all DSSP features [9, p. 29]. Hence, it is necessary that data sources can be extended on various ways. E.g. a source don't store its own meta-data, so there has to be an external meta-data repository for it.

A DSSP should also support updating data [9, p. 29]. Of course, the mutability of the relevant data sources determines the effects of updates. Other key services would be monitoring, event detection and the support for complex workflows (e.g. is desired that a calculation is done if new data arrives and that the result will be distributed over a set of data sources) [9, p. 29]. On a similar way a DSSP should support various forms of data mining and analysis. The user should be able to discover relevant datasources and to look for their completeness, correctness and recency [9, p. 29]. A DSSP indeed should recognise when data of the domain is incomplete.

3.1 Components and Services

In this section we will look at the components forming a fully-fledged dataspace and the services providing these components. Figure 3.1 shows the components as they were defined in [2]:

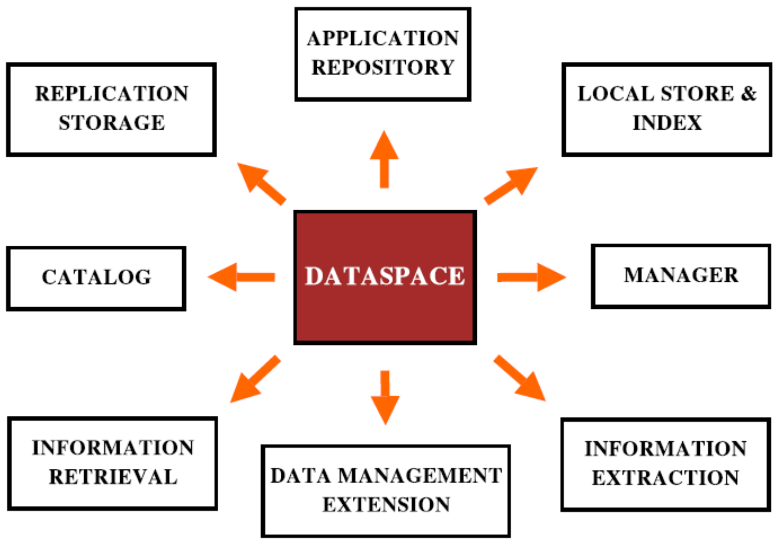


Figure 3.1: Environment of a Dataspace [2, p. 2]

Catalog: The most basic service a dataspace should provide is the cataloging of data elements of all participants. A catalog is an inventory of data resources containing all important information about every element (source, name, storage location inside the source, size, creation date, owner, etc.) of the dataspace. It is important that the catalog includes the schema of the source, statistics, rates of change, accuracy, completeness query answering capabilities, ownership, and access and privacy policies for each participant. Relationships may be stored as query transformations, dependency graphs, or even textual descriptions.The catalog is the infrastructure for the most other dataspace services. Search and query are two main services a DSSP must provide. A user should be able to state a search query and iteratively precise it, when appropriate, to a database-style query. For the dataspace approach it is a key tenet that search should be applicable to all of the contents, regardless of their formats. The search should include both, data and meta-data. Then, it can support basic browsing over the participant's inventories. The catalog may reference a meta-data repository to separate the basic and more detailed descriptions. It isn't a very scalable interface, but it can be used to response to questions about the presence or absence of a data element, or determine which participants have documents of a specific type. On top of the catalog the DSSP should have a model-management environment allowing the creation of new relationships and manipulation of existing ones.

Information Retrieval: The Information Retrieval component (in [20] stated as 'Search and Query') can be split up into queries and searches, two different methods of information retrieval, and represent together one of the main services a DSSP should support. Generally, queries and searches should be supported by all participants of the Dataspace, regardless of their used data model. It shouldn't make any difference for a user to operate on a sole database or on a Dataspace. A good known and simple search operation is keyword searching. The support of spanning such a search method over all participants is a challenging research topic: The development of methods for keyword searching on relational and XML databases was done by the data-engineering community [21, 22, 23]. For to support a global query functionality allowing to formulate uniform queries on all Dataspace participants, intelligent methods for interpreting and translating of queries in several languages are required. Methods for query translation were investigated by a large body of research communities [24, 25]. In the following are listed the main requirements having to be supported by this component:

Query everything: any data item should be queryable by the user regardless of the file format or data model. Keyword queries initially should be supported. When more information about a participant is collected, it should be possible to gradually support more sophisticated queries. The transition between keyword query, browsing and structured querying should be gracefully. Also, when answers are given to keyword (or structured) queries, the user should be able to refine the query through additional query interfaces.

Structured queries: queries similar to database ones should be supported on common interfaces (i.e. mediated schemas) that provide access to multiple sources, or can be posed on a specific data source (using its own schema). The intention is, that answers will be obtained also from other sources (as in peer-data management systems). Queries can be posed in a variety of languages (and underlying data models), and should be translated into other data models and schemas as best as possible with the use of exact and approximate semantic mappings.

Meta-data queries: it is of essential importance that the system supports a huge variety of meta-data queries. These include (a) source inclusion of an answer or how it was derived or computed, (b) time steps provision of the data items that are included in the computation of an answer, (c) specification of whether other data items may depend on on a particular data item and the ability to support hypothetical queries. A hypothetical question would be 'What would change if I removed data item X?' (d) Querying the sources and degree of uncertainty about the answers. Queries locating data, where the answers are data sources rather than specific data items, should be supported, too.

Monitoring: all stated Search and Query services should also be supported in an incremental form which is also applicable in real-time to streaming or modified data sources. It can be done either as a stateless process, in which data items are considered individually, or as a statefull process. In the latter multiple data items are considered.

Local Store and Index: This component is responsible for caching search and query results, so that certain queries can be answered without the need of accessing the actual data sources. Furthermore it supports the creation of queryable associations between the participants. The component should try to achieve the following goals:

- to create efficiently queryable associations between data items in different participants. Important is here, that the index should identify information across participants when certain tokens appear in multiple ones (in a sense, a generalization of a join index)
- to improve accesses to data items with limited access patterns. Here, the index has to be robust in the face of multiple references to real-world objects, e.g. different ways to refer to a company or person.
- to answer certain queries without accessing actual data sources. Thus, the query load is reduced on participants which cannot allow ad-hoc external queries.
- to support high availability and recovery

The index has to be highly adaptive to heterogeneous environments. It should take as input any token appearing in the dataspace and return the location at which the token appears and the roles at each occurrence. Occurrences could be a string in a text file, element in file path, a value in a database, element in a schema, or tag in a XML file.

Discovery Component: This component (not listed in Figure 3.1) locates participants in a dataspace, creates relationships between them, and helps administrators to refine and tighten these relationships. For each participant the component should perform an initial classification according to the participant's type and content. The system should provide an environment for semi-automatically creating relationships between existing participants and refining and maintaining existing ones. This involves both, finding which pairs of participants are likely to be related, and then proposing relationships which a human can verify and refine. The discovery component should also monitor the content in order to propose additional relationships in the dataspace over time.

Data Management Extensions: This component (in [20] stated as *Source Extension Component*) provides possibilities to improve low-level working dataspace components. All these base components have only limited data management capabilities. It is a task of a DSSP to provide additional functionality such as backup, recovery and replication. Some participants may not provide significant data management functions. For example, a participant might be no more than departmental document repository, perhaps with no other services than weekly backups. A DSSP should support to enrich such a participant with additional capabilities, such as a schema, a catalog, keyword search and update monitoring. It may be necessary to provide these extensions locally, as there might exist applications or workflows that assume the current formats or directory structures. This component also supports "value-added" information held by the DSSP, but not present in the initial participants. Such information can include "lexical crosswalks" between vocabularies, translation tables for coded values, classifications and ratings of documents, and annotations or linked attached data set or document contents. Such information must be able to span participants in order to link related data items.

Information Extraction: The world-wide web has become a large data container by now. For allowing to postprocess data obtained from the web, techniques for web information extraction, extracting relevant information from semi-structured web pages should be supported. For further processing extracted content has to be converted to structured data, and saved locally. Non structured web documents have first to be classified using text mining techniques, which organize the parsed documents into groups with the help of ontologies [26]. Based on these ontologies a keyword search is able to find individual documents. Structured documents allow easier access and integration due to the rich semantically information included in the data representation.

Manager: In order to realize the above mentioned functionality, a central component managing the system and interacting with the user is needed. Alongside user authentication, right assignments and other services, this manager component is responsible for communicating with all participants. Thus, this component serves as an interface between the users and the participants of the Dataspace.

Replication Storage: Allows to copy participant data in order to increase its access time. This results in high availability, and high recovery is supported.

Application Repository: With this component the user is able to share data analysis tools, domain specific models, evaluations, etc., which can be applied to the (available) data of the Dataspace.

3.2 Query answering

In a dataspace, queries might be posed in a wide range of different languages [20, p. 3]. But most of the activities will properly begin with a keyword search, but it might also be common to see queries as a result of a form (which results to queries with several selectable predicates). More complex queries arise when a user interacts deeper with a certain data source. If not stated explicitly, it can be assumed that a user likely believes a query considers all relevant data in a dataspace, regardless of the used data model or schema. Even if a query is posed to a data source, it is implicitly expected that the system considers the data of other sources, as well. If additional answers are desired, one has to do transformations on the schema and the data model.

Challenges of answer querying

Answers corresponded to queries of a dataspace are different from traditional queries in several ways. In the following we list challenges a dataspace has to deal with [20, p. 3-4]:

- **Ranking**: Queries are typically sorted by their relevance, similar to a web search engine. Ranking is necessary not only for keyword search, but also for structured queries, when transitions to other data sources should be approximated.
- **Heterogenity**: Answers will come from many sources and will differ in their used data model and schema. The Ranking has to manage heterogeneity, too.

- **Sources as answers**: In addition to base elements (e.g. documents or tuples), a DSSP should also be able to provide sources. This means that it returns links to locations where additional answers can be found.
- **Iterative queries**: Normally, the interaction with a dataspace can't be reduced to the process of posing a sole query and getting an answer to it. Instead, a user is involved in an information finding task that requires a sequence of queries, each being a refinement or modification on the previous ones.
- **Reflection**: It is expected, that a DSSP reflects on the completeness of its coverage and the accuracy of its answers.

3.3 Uncertainty in Dataspaces

In the paper "Data Modeling in Dataspace Support Systems" [3], the authors face the issue of uncertainty in dataspaces. In their view, a dataspace needs to model uncertainty in its core. In their work, they described the concepts of probabilistic mediated schemas and probabilistic mappings as enabling concepts for DSSPs. With this foundation, it is possible to completely bootstrap a pay-as-you-go integration system. In order to develop DSSPs, it is impossible to rely on the same data modeling paradigms data integration systems use. One cannot assume that the mediated schema is given in advance, and that the schema mappings between the sources and the mediated schema will be accurate. Therefore, the authors argue that uncertainty has to be considered in the mediated schema and in the schema mappings.

Now, we discuss the different kinds of uncertainty, that can arise in a dataspace [3, p. 123]:

- **Uncertain mediated schema**: the set of schema terms in which queries are posed is called the mediated schema. Not necessarily the set covers all the attributes in any source, it rather covers the aspects of the domain that the developer of the application wishes to expose to the user. For several reasons uncertainty happens in the mediated schema. First, if the mediated schema is derived from the data sources during bootstrapping, there will be some uncertainty about the results. When domains are broad, there will be also some uncertainty about how to model them, because in general there will be overlappings in different topics.
- **Uncertain schema mapping**: schema mapping defines the semantic relationships between the terms in the sources, and the terms used in the mediated schema. Although, schema mappings can be inaccurate. In a dataspace many of the initial schema mappings are probably automatically derived, and therefore they might be inaccurate. In general, it is impossible to create and maintain precise mappings between data sources.
- **Uncertain data**: some of the data may be obtained by automatism, as data sources may not always be structured well. Additionally, unreliable or inconsistent data may be contained in systems with many sources.
- **Uncertain queries**: probably much of the early interaction with a dataspace will be done through keyword queries, as the users aren't aware of a (non-existent) schema. The queries have to be translated into some structured

form so they can be reformulated with respect to the data sources. At this point multiple candidate structured queries could be generated, and thus some uncertainty arises about which query captures the real intention of the user.

The most fundamental characteristic of a dataspace system handling uncertainty is that it is based on a probabilistic data model. In contrast to a traditional data integration system which includes a single mediated schema and assumes a single (and correct) schema mapping between the mediated schema, and each source, the data integration module of a DSSP attaches possibilities to multiple tuples, mediated schemas, schema mappings, and possible interpretations of keyword queries posed to the system.

For a database it is assumed that queries are posed as keywords. This is contrary to traditional integration systems which assume the query to be posed in a structured fashion(i.e. that it can be translated to some subset of SQL). So, a DSSP must first reformulate a keyword query into a set of candidate structured queries before it can reformulate the query onto the schemas of the data sources. This step is also called keyword reformulation. It is important to note that keyword reformulation differs from keyword search techniques on structured data (see [21], [23]) in that (a) it does not assume access to all data in the sources, or that the sources support keyword search, and (b) tries to distinguish different structural elements in the query in order to pose more precise queries to the sources. In any case keyword reformulation should benefit from techniques that support answering search on structured data.

Different is also the query answering in a DSSP. It doesn't find necessarily all answers on a given query, rather than typically find the top-k answers, and rank these answers most effectively. The architecture of the proposed system is shown in figure 3.2.

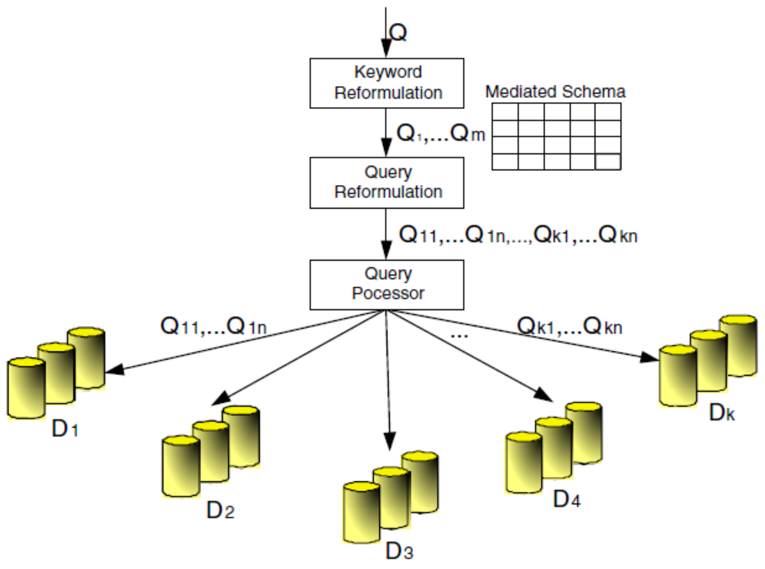


Figure 3.2: Architecture of a dataspace system that handles uncertainty [3, p. 124]

The DSSP contains a number of data sources and a mediated schema (probabilistic mediated schemas are omitted here). When the user poses a query, which can be either a structured or a keyword query, the systems returns a set of answer tuples, each with a calculated probability. If a keyword query was posed, a keyword reformulation has firstly be done to translate it into a set of candidate structured queries on the mediated schema. Otherwise, the candidate query is the posed query itself.

3.4 Multimedia Dataspaces

Research in the field of dataspaces is currently very active, but despite of the deep interest existing models suffer on a number of shortcomings limiting their applicability [10, p. 1]. These include the tendency of overlooking the different types of relations that can exist between data items, which restricts the amount of information they can generally integrate. Second, they focus on classical text data ignoring the specifics of multimedia data. Third, they don't provide the fine-granularity in the persistence of integrated data that is needed in certain domains.

To address these issues a representation model for dataspaces was developed, which sees the dataspace as a set of classes, objects (instances of classes) and relations [10, p. 1]. In the latter case exist relations between classes (CRC), relations between objects and classes (ORC) and relations between objects (ORO). Furthermore relations can be internal or external, and define whether the anticipating relation objects are in the same data source or in different ones.

A design goal of the model is the maximization of its expression in terms of the types of relations that it can represent, enabling it to deal with information originating from structured data, semi-structured data, ontologies and other forms of knowledge representation as well as from canonical knowledge.

Important to note is the fact that the model includes *similarity relations* in the type of relations [10, p. 3-4]. This property makes this model also suitable for integrating multimedia data. Similarity functions are a feature of multimedia data that defines a measurement for non-exact matching between objects. This kind of relations can be used to derive relations between other objects in the dataspace.

Additionally the model introduces the concept of *dataspace views* [10, p. 3]. This makes it possible to store query results of an existing dataspace into a new sub-dataspace with different modes of persistence (virtualized view, materialized view, mode of synchronization with the content of the original sources,...).

The multimedia dataspace model was implemented in [27, p. 30] using RDF as its canonical data model.

4 State-Of-The-Art projects

In this chapter we will analyze related data integration and dataspace works in the healthcare sector. This will help us to properly design and implement our own system.

We will cover the following works:

- DebugIT which has a mediator-based architecture and uses ontology-based data integration.
- 'Dataspace Integration in medical research', the doctor thesis of Sebastian H.R. Wurst

4.1 DebugIT

The DebugIT project is a good reference for analyzing how medical data integration can be done, though this project didn't use a dataspace approach but an ontology-mediated approach [4, 1]. This means DebugIT uses a mediator-based approach as archtecture and ontologies for semantic data integration.

DebugIT stands for 'Detecting and Eleminating Bacteria Using Information Technology', and its main goal is to provide a platform for high-throughput analysis of distributed clinical data as a response to the spread of antibiotic resistance of infectious pathogens in European hospitals[28].

The team of the DebugIT project released the architecture of their system in [29]. In this paper is stated that the system uses an ontology-based approach for allowing antibiotics resistance data being semantically and geographically interoperable. This makes it possible to integrate distributed clinical data EU-wide in real-time for monitoring antibiotic resistance. As well, the system is structured in three tiers and has a service-oriented architecture (SOA). Figure 4.1 shows the layered architecture of the system of DebugIT.

Altogether there are four data representation layers marked with the Roman Numerals I-IV. The query flow between the data layers is specified with 1-3 and the corresponding mappings are given with the Greek letters α, β, and γ.

On the first integration layer (**I**) relational data are lexical normalized by the use of mappings of medical terminology and morphosemantic mapping employed by the Averbis Morphosaurus software [30]. Ambiguities can be resolved with ontological expressions (formulated in OWL) on the integration layers II and III.

The layer **II** works with RDF data, wherefore relational data from the layer I is transformed to RDF through D2R mapping calls [1] on the *first mediation layer* (1).

[1] `http://d2rq.org/`

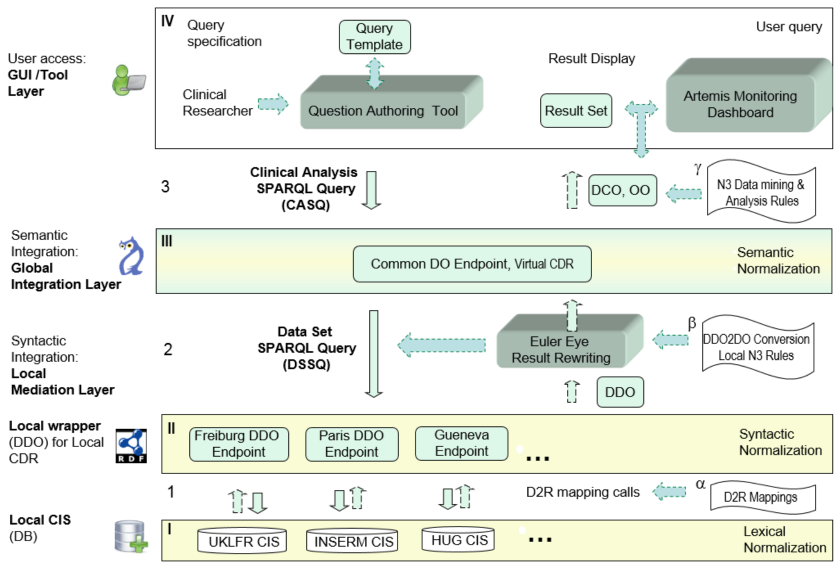

User access: GUI /Tool Layer
IV
Query specification
Query Template
User query
Result Display
Clinical Researcher
Question Authoring Tool
Result Set
Artemis Monitoring Dashboard
3
Clinical Analysis SPARQL Query (CASQ)
DCO, OO
γ
N3 Data mining & Analysis Rules
Semantic Integration: Global Integration Layer
III
Common DO Endpoint, Virtual CDR
Semantic Normalization
Syntactic Integration: Local Mediation Layer
2
Data Set SPARQL Query (DSSQ)
Euler Eye Result Rewriting
β
DDO2DO Conversion Local N3 Rules
DDO
Local wrapper (DDO) for Local CDR
II
Freiburg DDO Endpoint
Paris DDO Endpoint
Gueneva Endpoint
...
Syntactic Normalization
1
D2R mapping calls
α
D2R Mappings
Local CIS (DB)
I
UKLFR CIS
INSERM CIS
HUG CIS
...
Lexical Normalization

Figure 4.1: The layered mediator architecture of the DebugIT project

On the second integration layer information about the data and their corresponding vocabulary is stored in Data Definition Ontologies (DDOs) [31]. A DDO bridges a local data model and a semi-formal data model on the local mediation layer for integrating syntactic data and provides an Extract, Transform, Load (ETL)-process [1, p. 382] for the next integration layer. So materialized data integration is partially done on this layer. For each Clinical Data Repository (CDR) such a DDO is locally created.

A CDR integrates several (local) data sources to provide up-to-date patient informations to clinicians in real-time [32, p. 82]. CDRs deliver data for only one patient at a time.

Layer II contains also a SPARQL endpoint, for allowing Layer III to query its data. These queries are specified through a Data Set SPARQL Query (DSSQ).

On the *second mediation layer* (2), in the figure called *local mediation layer*, the local DDO data is bind to the global DebugIT Core Ontology (DCO) [33], which is the ontology of layer III. The corresponding mapping is done through DDO2DCO using the N3 language [2] and Simple Knowledge Organization Structure (SKOS) mappings [3]. The schema mapping is done by the Euler Eye Reasoner [4], which also creates implicit knowledge using logical inference.

Data layer **III** represents a virtual Clinical Data Repository (vCDR) which joins the local Clinical Information Systems (CISs). Important to know is that in the vCDR are now fully (semantically) integrated, and as the name implies, a vCDR *virtually* integrates the data, which means it is not duplicated anymore. Through the virtualization data of all CIS can be queried globally. Also privacy issues are handled properly, since the data is not stored outside from the CISs. Also on layer III

[2]http://www.w3.org/TeamSubmission/n3/
[3]http://www.w3.org/TR/2009/REC-skos-reference-20090818/
[4]http://eulersharp.sourceforge.net/

clinical analyses can be performed over Clinical Analysis SPARQL Queries (CASQ (3)).

On the last layer (**IV**) a user or a monitoring tool can query the integrated clinical data.

To summarize, the mapping is performed iteratively in a stack-like manner: The first mapping α(D2R mappings) transforms the relational database layer (I) to the RDF representation layer (II). The next mapping β(N3 and SKOS) transforms the RDF layer II to the Domain Ontology (OO) layer III, where the data is globally queryable as CASQ over mapping γ(DCO and OO).

4.2 Dataspace Integration in medical research

'Dataspace Integration in medical research' (original title 'Dataspace Integration in der medizinischen Forschung') is a German doctor thesis written by Sebastian H.R. Wurst, and submitted in 2010 [4]. In his thesis Wurst evaluated the dataspace concept for medical research and designed a software architecture that could be used for dataspaces. He also designed a generic data model expressed in RDF and implemented a framework for agile software development. In his conclusion he states the dataspace concept indeed is suitable for the health-care sector, since it is a heterogeneous environment that is constantly changing and has dynamically adept to these changes. For these reasons a dataspace is more suitable for health-care and medical research than a classic data integration approach.

In this section we want to analyze how Wurst defined the software architecture for a DDSP. Figure 4.2 shows an overview of the software architecture:

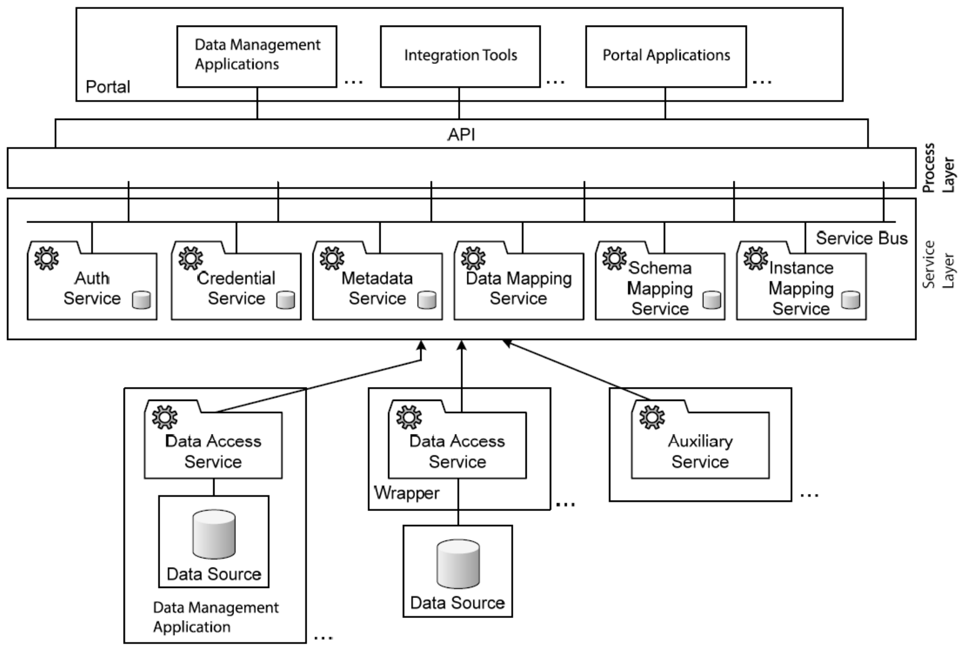


Figure 4.2: Overview of the software architecture of a DSSP used in the doctor thesis of Sebastian H.R. Wurst [4, p. 117, Figure 31, english translation]

The architecture is subdivided into a service, process, and application layer:

- **Service layer:** Provides services to the underlying datasources. E.g. the authentication service provides functionality for authentication, authorization, and session management. The credential service manages known data sources with autonomous user administration and stores data for user account specific authentication. The meta-data service allows access to schema and meta-data of the data sources.
- **Process layer:** Coordinates interactions between the services and thus implements the application logic. This layer provides interfaces so that the application layer can use it.
- **Application layer:** Provides applications that operate on top of the process layer.

Applications can be subdivided into three categories:

- **Data management applications:** Use the DSSP for data integration
- **Integration tools:** Used for doing tighter and incremental data integration.
- **Portal applications:** Allow access to services providing additional benefits for a specific integration solution. They don't manage own data. An example for this kind of applications would be statistics that should be applied on top of integrated data.

The DSSP has a canonical data model and Wurst used RDF for it. The communication between the services and the data sources is done through so-called *data access services*, that essentially are wrappers for a specific data source. Auxiliary services for accessing the data sources are used, too, but play a subordinary role.

Interestingly enough, Wurst's DSSP architecture resembles the mediator-based data integration solution used by DebugIT: both architectures use wrappers for accessing the data sources that are rather lightweight. The wrappers don't need to know any integration system and thus autonomy of the data sources is preserved. The mediator-based architecture uses mediators to implement services on top of the data sources, and applications can use the mediators to implement a data integration solution. This is also similarly done in Wurst's architecture: The process layer (and thus indirectly the service layer) that is used by the application layer doing concrete data integration. And both systems use RDF as a canonical data model and ontologies for semantic data integration.

Of course the architectures have a fundamental difference: As all data integration solution the applications of a mediator-based architecture operate on semantically integrated data, whereas a DSSP allows data co-existence and incremental data integration. However, it is remarkable how similar the structures of the two architectures is. We reuse these fundamental similarities for our own system, as it seems to be proven.

5 MeDSpace

In this chapter we will present the developed system MeDSpace (*abbr.* for Medical Dataspace), whereas in the next chapter we will look at more implementation specific details.

The purpose of MeDSpace is to provide a distributed test environment of medical datasources. Thereby, the focus is the provision of a keyword search functionality over medical datasources. As distributed keyword search is the foundation for every Dataspace system, MeDSpace can be used as a starting point for a much more evolved dataspace, or as a framework for implementing advanced dataspace functionality. Although the system uses medical datasources for its test data, the system is general enough to be used for other dataspace-related projects or systems that need keyword search functionality over a set of multiple heterogenous (multimedia) datasources.

In figure 5.1 the overview of the MeDSpace system is illustrated.

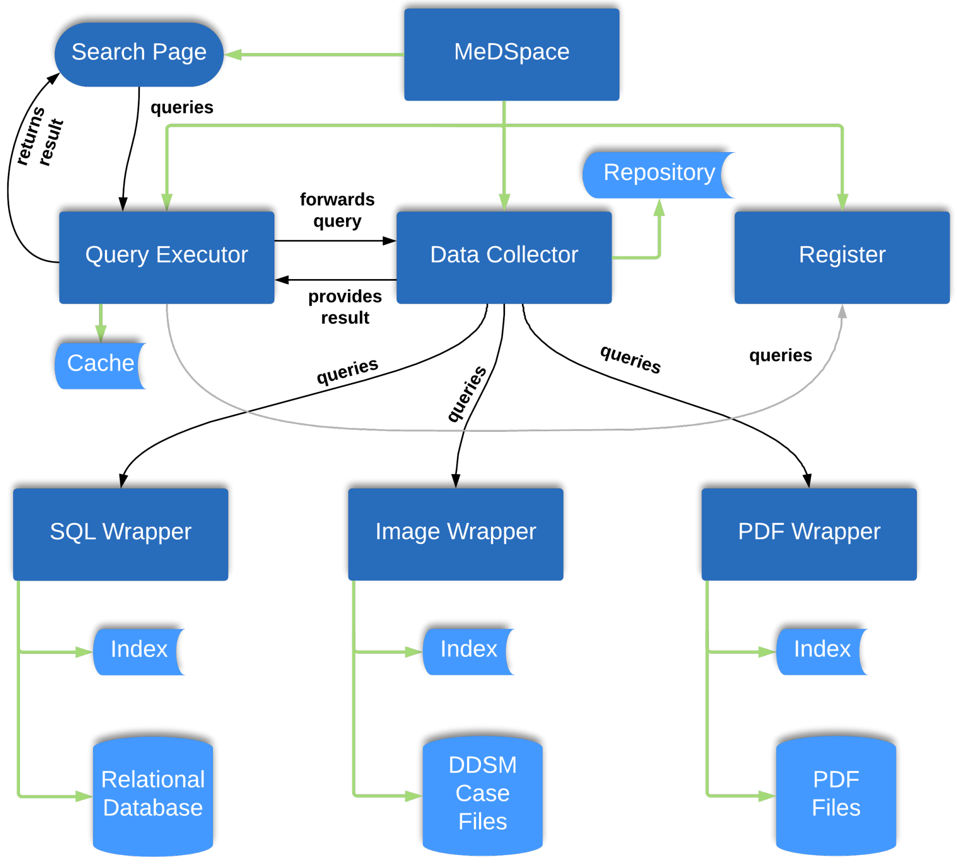


Figure 5.1: MeDSpace Project overview

The system can roughly be divided into a *wrapper* and a *global MeDSpace* category: The *wrapper* category includes everything from managing the data of a specific data-sources (e.g. from a relational database). Whereas the *global MeDSpace* category comprises datasource management and services on a global view. The modules belonging to this category are the *Register*, the *Data Collector*, the *Query Executor*, and the *Search Page*.

Now, we want to look at the requirements met by MeDSpace. Therefore we look at the purposes of each module of the system.

5.1 General

MeDSpace uses RDF [34] as its canonical data model. Although a dataspace system is not restricted to use a specific data model, it simplifies the complexity of data management. Furthermore RDF is proposed to be used for Linked Data [35] in the Semantic Web field. Although a different research field, both fields face semantic data-integration.

All wrappers and the register have configuration files. These configuration files use XML, and each configuration file has a corresponding *XML Schema Definition* (XSD)[14] file, that is used to validate the configuration file. This allowed us to generate automatically Java classes out of the XSD files using *Java Architecture for XML Binding* (JAXB) [36].

5.1.1 Importance of unicode

A dataspace includes data sources possibly spread around the globe. It is near inferring to support a wide area of different languages. In terms of character sets, it is therefore necessary to use a unicode encoding. Unicode is a system assigning each character a unique code point and is designed to support the worldwide interchange, processing and display of texts written in different languages[37].
There exist several encodings for unicode. The better known are are the UTF and UCS encoding families. In the draft of HTML5 is advised to use UTF-8 for new web pages[38]. Thus, to simplify processing we follow the recommendation and use UTF-8 throughout MeDSpace. If a data source doesn't use UTF-8, it is the task of its wrapper to do a proper conversion to UTF-8.

5.1.2 Keyword query language convention

As already previously told, the one service that every wrapper has to implement is the keyword search functionality. In order to search for specific keywords, it is important to define a convention how the keyword search should work, since there are several possibilities. E.g. if a query is posed with two keywords, should a query be created for all results that contains both keywords, or is it enough if one of the keywords is contained? Should boolean operators be allowed (AND, OR, NOT)?

The design decision is at follows: Rudimentary keyword search functionality suffices and several keyword searches have to be interpreted in an 'AND like' manner by default, i.e. all of the stated keywords have to occur to fulfill the query condition. Additionally the user should be able to use an 'OR' operator instead. This option is provided by the search page.

The reasons are: no complex global query language is necessary that all wrappers have to implement, and this leads to much more flexible implementation decisions. We think that it is more likely for a user to achieve results including all stated keywords and not only one of them. So we decided to use the 'AND' operator by default. In certain cases the use of the 'OR' operator might be helpful, so we added it, too. As wrappers can provide as many services as wanted, it is easy to provide a much more powerful keyword search service. So this design decision doesn't restrict the power of a wrapper.

5.2 Wrappers

Each local datasource has its own wrapper. The wrapper is similar to the wrappers used in a mediator-based system, but provides dataspace specific functionality: the wrapper...

- manages the data exchange between the datasource and extern MeDSpace services.
- provides keyword search functionality (if the data source does not already provide it itself). Therefore it creates an index of the data of the wrapped datasource and maintains it.
- converts search results from the datasource to the canonical data model.
- registers and deregisters the datasource from MeDSpace by communicating with the *Register* module. The wrapper informs the register about the services the datasource and the wrapper provide.
- overcomes technical, syntactic and data model heterogeneity.

Thereby the *SQL Wrapper* is responsible for wrapping a relational database containing data created with the *Patient Data Generation Framework* (PDGF) tool created by Schmiedbauer [39]. The *Image Wrapper* provides access to case files of *Digital Database for Screening Mammography* (DDSM)[40] and the *PDF Wrapper* maintains a set of pdf files that were created specific for this system: the pdf files contain data based on Schmidbauer's test data.

5.2.1 File downloading service

Image Wrapper and PDF Wrapper both provide access to data coming from files. We decided that these wrappers add source URLs to the exported data. Additionally theses wrappers function as file servers so that it is possible that MeDSpace clients are able to access the source files from the exported RDF data. This is useful for datasources that have unstructured data.

5.3 MeDSpace server

We decided to create a server application that integrates global functionality of the MeDSpace system. This functionality is separated into modules. Our design goal was to design the modules in such a way that the global MedSpace server can be distributed in future iterations. Therefore the modules have to communicate by using REST services. In the following we will present this modules.

5.3.1 Register

The Register holds a list of active datasources that can be queried. Therefore it provides functionality so that wrappers can register and deregister a datasource. In order to know what services are provided by each datasource, the register also keeps records of these services. This allows other modules to use services of any datasource.

5.3.2 Query Executor

The task of the *Query Executor* is to accept a given keyword search query, transform this query to a service call, and instructs the *Data Collector* module to call the keyword search service for each registered datasource. To know which datasources are registered, this module communicates with the *Register* module. After the *Data Collector* has collected the query results of all datasources the *Query Executor* adds the search query to its query cache and returns the collected search result to the caller who requested the *Query Executor*. On the next request the cached query result will be returned without querying the datasources if no 'cache miss' has been occurred.

Note: It is not the task of the *Query Executor* to actually collect the search result from the datasources. That is done by the *Data Collector*. It does only instruct the *Data Collector* to do the collecting.

5.3.3 Data Collector

The *Data Collector* provides services that allow to query a specific datasource and store the query result into a specific RDF repository which is maintained by this module. Furthermore the *Data Collector* allows other modules to create and remove a repository that is coupled with a specific search query. This allows the *Query Executor* to merge the search results of all datasources into a seperate RDF repository. Then, again the *Query Executor* is able to delete the repository if the associated and cached query result should be deleted.

5.3.4 Search Page

The Search Page represents the *graphical user interface* (GUI) so that a user can easily interact with the MeDSpace system using a common browser. It provides a page for stating and sending keyword searches, allows the user to display the search result in the browser or download it as a file. Furthermore the search page allows the user to inspect the list of the current registered datasources and to delete the query cache.

6 Implementation

In this chapter we focus on implementation details of the MeDSpace system:

- the used technology
- the implementation structure
- reasons for using this kind of implementation structure
- details explained for users and developers to use the system correctly.

6.1 Used technologies

In this section we want to present the technologies used in MeDSpace and explain for each technology the areas of application.

6.1.1 Programming language - Java

MeDSpace is written completely in Java 8. The reason for java are diverse:

- its platform independence is very helpful in a heterogeneous environment
- Not reaching the performance of C++, Java with its JIT compiler has decent performance [41, p. 425]. Even though performance isn't the most important, it should not be neglected, as the system potentially has to handle very large data sets
- last but not least, Java has great library support for Web development

Of course, other languages could be used, too, but Java definitively is a good choice.

6.1.2 Resource Description Framework

The *Resource Description Framework* (RDF) [34] is used as the canonical data model in MeDSpace. Figure 6.1 shows how rdf data is structured. RDF is a graph-based abstract data model. With abstract we mean it is a conceptual data model, so RDF says nothing about serializing data. But there exists plenty serialization formats that can be used to serialize RDF data. A popular RDF serialization format is Turtle [5] that is known to be easily human readable. It facilitates merging data expressed in different schemas, as objects are identified by URIs, and with the help of ontologies semantic data integration can be performed by logical inference.

[5]`https://www.w3.org/TR/turtle/`

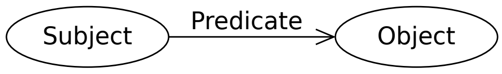


Figure 6.1: A predicate connects two nodes (Subject, Object) forming an RDF Triple [6]

RDF data consists of resources which are the nodes of the rdf graph, and predicates which are directed edges between the resources. So RDF is a directed graph.

Resources can be IRIs, literals, and blank nodes. Resources with an IRI are identified through its IRI. Two resources with the same IRI are supposed to be equal. Blank nodes are also called anonymous resources, as they specify the existence of an abstract thing, but it is not identifiable. So blank nodes are always unique. Literals are used for values like numbers, strings or a date, but aren't identifiable, too. Predicates are always IRIs and can also be resources. They are also identified by their IRIs.

The idea of RDF is to represent data in form of triple statements that consists of two resources (subject and object in figure 6.1) and a predicate. The statement 'A has a property B' could be expressed in RDF with two resources, A (the subject) and B (the object) and a predicate with an IRI that should form the 'has' predicate. It should be noted that a subject can be an IRI or a blank node, but not a literal. As a result no statements can be made about a literal.

Listing 6.1 shows an example of how rdf data can look like using Turtle syntax. In line 3 we can see the aforementioned example statement 'A has B'. An interesting property of Turtle is the ability to define prefixes for namespaces, as done at line 1. Prefixes are used to shorten IRIs, thus increasing readability. Thus, the statement in line 3 and the statement in line 5 are equal: in line 3 the fully qualified IRIs are used, whereas in line 5 the 'ex' namespace prefix is used.

Line 7 shows an example of a statement with a literal value.

Line 8 and 9 shows two different ways of blank node usage: In line 8 a so-called 'labeled' blank node is created. This allows a developer to give a blank node same properties by using the labeled like an IRI. A property is stated for this labeled blank node at line 11. But it should be noted that a blank node is not visible from the outside (i.e. in a SPARQL query). In line 9 the usage of an 'unlabeled' blank node is illustrated: everything that belongs to the blank node is defined inside the rectangular brackets.

```
1  @prefix ex: <http://example.org/#>
2
3  <http://example.org/#A>
4          <http://example.org/#has> <http://example.org/#B> .
5
6  ex:A ex:has ex:B .
7  ex:A ex:hasLiteral 20 .
8  ex:A ex:hasLabeledBlankNode _:blank .
9  ex:A ex:hasUnlabeledBlankNode [ ex:hasLiteral 5 ] .
10
```

[6]https://www.w3.org/TR/rdf11-concepts/rdf-graph.svg

```
_:blank ex:hasLiteral 5 .
```

Listing 6.1: RDF example data in Turtle syntax

In MeDSpace we used the RDF4J framework (version 2.2.2) [42] that enabled us to read and write RDF content from within Java. Originally we planned to use Apache Jena [43], but it wasn't possible for us to convert the RDF data into appropriate input and output stream classes without the Jena PipedRDFIterator class [7], that unfortunately creates a new thread for processing the rdf triples. But that isn't an option for us, as this solution doesn't scale well. MeDSpace uses its own interfaces for handling rdf data in order to be independent from any third party framework. Thus it would be relatively easy to support a different rdf framework. To do so, one can implement a custom version of the *de.unipassau.medspace.rdf.RDFProvider* interface.

6.1.3 Apache Lucene

Apache Lucene is a full-featured text search engine [44] at version 6.6.6. In MeDSpace it is used to implement the keyword search functionality and for the search indexes used by the wrappers.

6.1.4 Play Framework

Play is a framework for web development and written in Scala, but it provides Java bindings [45]. So it is possible to use Play without any complications. In MeDSpace we used Play in its version 2.6.6. We utilized the framework for implementing the REST services and for the GUI.

Play has its own default folder structure. In order to understand how the modules are structured, it is necessary to have at least a rough understanding of it:

- **app** In this folder all the source code is stored. This includes also html templates that are used to generate dynamic html pages. These templates have to be stored in the subfolder *views*. The same applies to controllers which have to be stored inside the *controllers* subpackage.
- **conf** Contains configuration files. The file *application.conf* is used to configure the Play framework (e.g. the port number for the http server should run on). The *logback.xml* file is the configuration file for the logging backend logback [8] and the file *routes* contains the definition of relative URL to java method mappings. This is used to define on which URL a service is located. As some modules call services by its URL name, the URLs routes shouldn't be modified carelessly. For example, if you change the relative URL for registering a datasource, all wrappers have to be updated to use the new URL.
- **public** Resource folder for the http server. Here is stored static content that should be transferred to a client, like javascript files or css files.

[7] https://jena.apache.org/documentation/javadoc/arq/org/apache/jena/riot/lang/PipedRDFIterator.html

[8] https://logback.qos.ch/

- **project** Contains additional build configurations. E.g. we used the sbt-sass plugin to compile sass to a css file.

Play needs the Simple Build Tool [46](SBT). As a consequence we had to use SBT, too. Thus we used SBT as build tool for the whole project. If you just want to run MeDSpace, SBT is not required, but if you want to compile it you must have installed it at least on version 0.13.15 .

The play framework supports dependency injection powered by Google's Guice [9]. This way it is possible to inject dependencies by defining a special class that is instantiated by Play on startup. The definition of this class is done in the *application.conf* file by assigning the property *play.modules.enabled.* In MeDSpace this functionality is used to inject global class instances, e.g. for providing access to the configuration files or access to rdf processing methods.

6.1.5 JAXB

With the Java Architecture for XML Binding [36] (JAXB) we mapped the XML configuration files to auto-generated Java classes. This simplified the configuration reading, reduced programming faults, and enabled us to do changes faster. If you want to generate the java classes by yourself, you have to look at the subfolder *jaxb-generation* of your target module (e.g. SqlWrapper). There you'll find batch and shell scripts you just have to execute. The only requirement is that the *xjc* binary is findable through the environment variable of your operating system. This should be the case if you properly have installed the JDK.

All xjc generation commands are structured as follows:

xjc -b 'binding-file' **-p** 'target-package' **-d** 'output-directory' **-extension** 'target-xsd-file'

The *binding-file* is used to customize the generated java classes, e.g. the class can be assigned a different class name, *output-directory* specifies the directory where the generated java classes should be saved to, and *target-xsd-file* is the xsd file to use. The xsd file is the specification how the XML configuration should look like and therefore from the xsd file the java classes can be derived from.

In listing 6.2 you can see an extract of *medspace-d2rmap-binding.xml*, where a complex type *Bridge* from the *Medspace_D2Rmap.xsd* xsd file is mapped to a class called *BridgeParsing*:

```
<jxb:bindings
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:jxb="http://java.sun.com/xml/ns/jaxb"
    version="2.1">

    <jxb:bindings schemaLocation="Medspace_D2Rmap.xsd">
        <jxb:bindings node="//xsd:complexType[@name='Bridge']">
            <jxb:class name="BridgeParsing"/>
        </jxb:bindings>
    </jxb:bindings>
```

[9]https://github.com/google/guice

```
</jxb:bindings>
```

Listing 6.2: JAXB binding example

6.2 Wrappers - General

This section targets to present implementation details that is used by all wrapper implementations.

Each wrapper has a *general-wrapper-config.xml* configuration file that is located in the *bin/medspace* subfolder of the respective wrapper. This configuration file has to follow a specific structure that is defined in *medspace-wrapper-config-specification.xsd*. This XSD file is used to validate the general wrapper configuration file. The XSD file is packaged in the jar libraries of the wrappers, but the it can be found as a resource file in the *commons* package:
{medspace-root-folder}/project-sources/commons/src/main/resources

The following items can be configured through the general wrapper configuration file:

- **Services**: Specifies which services the wrapper supports. The name of the service is thereby the relative path for calling the service. MeDSpace translate this then to an absolute URL by using the wrapper host address, the url would be *{wrapper-address}/{service-name}*.

  Example: if the wrapper address is *http://localhost:9300* and the service name is *keyword-search*, then the resulting absolute URL would be:
  http://localhost:9300/keyword-search.

  It should be noted that MeDSpace expects the services in **lower case**. So you shouldn't define service URLs that use upper case characters, as URLs are **case sensitive**. Furthermore is expected from every wrapper to support the keyword-search service. So it is not explicitly necessary to state it in the configuration file.
- **IndexDirectoy**: The directory where index data should be stored. If the directory already contains indexed data, the wrapper will use it.
- **Namespaces**: Defines prefixes for rdf namespaces.
- **OutputFormat**: Specifies the RDF syntax, the wrapper should use.
- **RegisterUrl**: The URL to the register.
- **ConnectToRegister**: Specifies whether the wrapper should register itself on startup. This is primarily for testing and debugging.

6.2.1 Indexing

On startup the wrapper tries to fetch and index the data from the datasource for the keyword search functionality. If the index exists already (e.g. from a previous run), the wrapper won't reindex the data. But the wrapper provides a service for reindexing the data on runtime. The service can be found at */reindex*, uses no argument and http *GET*.

6.2.2 Keyword search service

Each wrapper supports keyword search functionality. The corresponding service can be found under **/keyword-search**, uses http GET, and needs the following arguments:

- **keywords**: A list of keywords to search for. The keywords have to be separated by spaces or commas.
- **useOr**: Specifies if the OR operator should be used instead of AND. This argument is optional. Its default value is *false*.
- **attach**: Specifies whether the result should be interpreted as a file attachment. This opetion is only relevant for browser. This argument is optional. Its default value is *false*.

6.2.3 Starting a wrapper

Before the wrappers are started the *medspace* server should be started. Each wrapper has start scripts for Windows and Unix systems that can be found at
{medspace-root-folder}/{wrapper}/bin . It is recommended to use them so that all gets properly configured (library paths, etc.).

6.3 SQL Wrapper

In this section we will talk about implementation details of the SQL wrapper. The task of the SQL Wrapper is to convert the relational data into RDF and providing a keyword search functionality, as SQL databases don't provide this kind of functionality. We used a MySQL datasource, but as several other relational databse vendors exist, one additional design goal was to produce a wrapper that can be used with any other database that uses SQL for querying. That will considerably reduce prospective maintenance work. Thus, a SQL wrapper was implemented, that doesn't use any vendor specific features.

It should be noted that a suitable JDBC driver has to be in the library directory of the SQL Wrapper, but it is not necessary to recompile the wrapper. Out of the box is supported MySQL. If other vendors should be supported, the user is encouraged to add the JDBC library to *{medspace-root-folder}/SqlWrapper/lib*. It will automatically be added to the classpath by the start script.

Before discussing how to perform the conversion from SQL to RDF, we want to look at the proper setup of the MySQL database.

6.3.1 Setting up a MySQL data source

To setup a MySQL datasource get a recent stable MySQL community server[10] and install it for your target platform. Additionally you will need the Connector/J components, the official JDBC driver for MySQL.

[10]`https://www.mysql.de/downloads/`

At time of writing the most recent stable versions are the community server 5.7.21 and the Connector/J 5.1.46. These versions are used for the thesis project and all following commands are related on them. If you're using different versions, assure that the instructions are adapted properly. As a detailed installation instruction for all supported platforms would break the mold, the reader is encouraged to consult the official manual[11]. Assure that the MySQL binary folder is integrated into your class path, so that you can access it globally in a shell/command line. Although not necessary, it is recommended for security reasons to set a password for the root user[12]. After installing the server, do postinstallation setup and testing[13].

To support UTF-8 add the following settings to your my.cnf (or my.ini)[14]

```
[client]
    default-character-set = utf8mb4
[mysql]
    default-character-set = utf8mb4
[mysqld]
    skip-character-set-client-handshake
    collation-server= utf8mb4_unicode_ci
    character-set-server= utf8mb4
```

Note that we use the character set *utf8mb4* instead of *utf8*. The reason is that we default *utf8* character set in MySQL actually uses only up to 3 bytes. Thus this character set represents only a subset of unicode. For more information about this topic we encourage you to read a blog post[15] of Mathias Bynens, a developer from Google and a specialist for Unicode.

Then restart the mysqld daemon. In the following it is assumed that you have a running MySQL server now that can be accessed via shell/command line. Before you connect to the MySQL server, you should assure that the application you use for connecting is using UTF-8 for user input and sending statements. So, validate that your shell/command line is using UTF-8. E.g. on windows system (before Windows 10) the command line isn't using UTF-8 by default [16]. Now try to connect to the database as the user root:

```
mysql -u root -p
```

If you've done all right, you should be connected to the database after entering and confirming the password that you've previously stated for the user root.

[11] `https://dev.mysql.com/doc/`

[12] `https://dev.mysql.com/doc/refman/5.7/en/default-privileges.html` , `https://dev.mysql.com/doc/refman/5.7/en/resetting-permissions.html`

[13] `https://dev.mysql.com/doc/refman/5.7/en/postinstallation.html`

[14] More informations about option files: `https://dev.mysql.com/doc/refman/5.7/en/option-files.html`

[15] `https://mathiasbynens.be/notes/mysql-utf8mb4`

[16] To set the encoding to UTF-8 on the windows command line change the active code page to 65001 and set 'Lucida Console' as the displaying font. In contrast to the font the code page is only active for the current console session. But you can automate this command with a AutoRun setting. For more information see `https://blogs.msdn.microsoft.com/oldnewthing/20071121-00/?p=24433`

The next step is to validate that UTF-8 is indeed continuously used. Execute:

```
SHOW VARIABLES WHERE Variable_name LIKE 'character\_set\_%'
OR Variable_name LIKE 'collation%';
```

The output should be as follows:

```
+--------------------------+--------------------+
| Variable_name            | Value              |
+--------------------------+--------------------+
| character_set_client     | utf8mb4            |
| character_set_connection | utf8mb4            |
| character_set_database   | utf8mb4            |
| character_set_filesystem | binary             |
| character_set_results    | utf8mb4            |
| character_set_server     | utf8mb4            |
| character_set_system     | utf8               |
| collation_connection     | utf8mb4_unicode_ci |
| collation_database       | utf8mb4_unicode_ci |
| collation_server         | utf8mb4_unicode_ci |
+--------------------------+--------------------+
10 rows in set (0.00 sec)
```

Listing 6.3: Expected character set output

Note that all variables except *character_set_filesystem* and *character_set_system* are set to use *utf8mb4*. Basically these variables are used to interpret and write data consistently in UTF-8. More information about the stated variables can be found on the manual [17].

The next step is to initialize the data source with a database and some content. Further we need a user which is used by the wrapper to communicate with the data source. The wrapper needs no writing rights and indeed we don't want it to change the data, so following the security rule 'As few rights as possible' we grant this user only reading rights for fetching data. The commands for initializing the data source and creating a read-user are in the file **init_mysql.sql** which is located in the appendix data in the folder **SqlWrapper/SQL/MySQL**.

To execute commands from a file execute while logged in as the root user:

```
SOURCE path_to_sql_file;
```

where *path_to_sql_file* is the full (absolute or relative) path to the sql file. Ensure that you **don't** have created a **medspace** database by yourself since *init_mysql.sql* will **drop** (delete) the database. The created read-user will be called **medspace_client**. If you want to edit the init_mysql.sql file, beware that the file is encoded in UTF-8. As we instructed mysql to use UTF-8 in every case, this encoding is required. Assure that your file editor saves the file in that encoding, too. Otherwise, it could come to conversion errors.

[17] `https://dev.mysql.com/doc/refman/5.7/en/server-system-variables.html#sysvar_character_set_client`

6.3.2 Startup

On startup the SQL Wrapper connects to the MySQL database. Therefore, the wrapper uses the JDBC URL that has been specified in *medspace-d2r-mapping.xml*. By default the port *3306* is stated in that file. If you use a different port you have to change it. If the wrapper is unable to connect to the database, it tries to reconnect two times. If no try succeeds, the wrapper shutdowns automatically.

6.3.3 D2RMap

In this sub section we want to look at the conversion from SQL to RDF. To do this the wrapper implements a specialized version of the D2rMap language. D2rMap was designed by Chris Bizer and is a declarative language to describe mappings between relational databases schemata and OWL/RDFS ontologies[5].

D2rMap is a general purpose language to export any SQL data to RDF. To better suit the needs for a dataspace wrapper the language was changed. The custom language is called *MeDSpace D2RMap* and its language specification is defined in *MeDSpace_D2RMap_Language_Specification.pdf* which is shipped with MeDSpace. But in general it is very similar to the original D2rMap.

The mapping is done as follows: At first the user specifies mappings in a config file. The mapping process is visualized in figure 6.2.

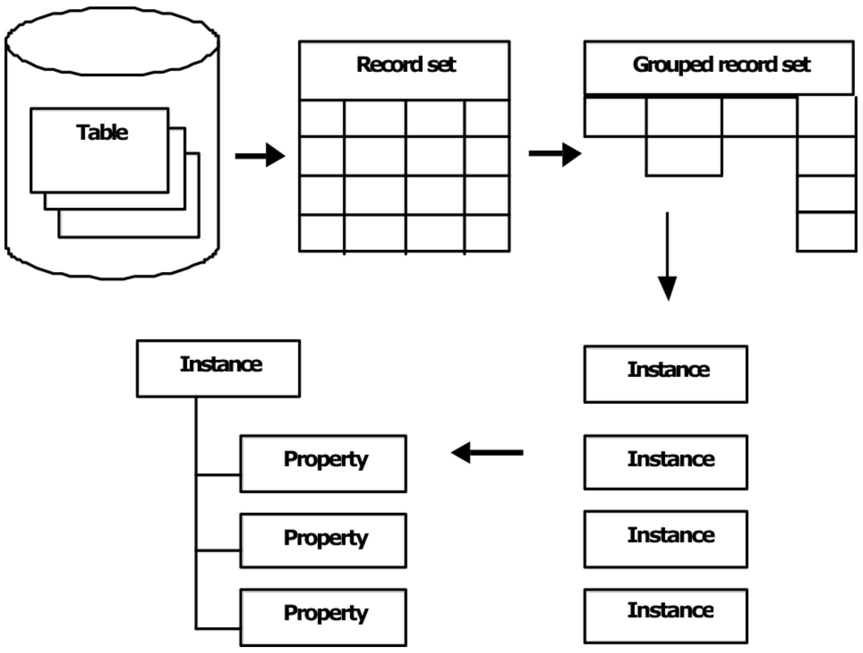


Figure 6.2: The D2r mapping process [5]

Each mapping is used to create RDF instances of a certain type. The mapping contains a SQL query that represents all the necessary data for creating the instances. Furthermore, in the mapping are columns specified that are used to create unique

IDs for the created RDF instances. The next step is to fetch the SQL data and group the record set according to the fore mentioned columns. Now, each row of the grouped record set represents a RDF instance, so the instances can be created by proceeding all rows. The last step is the creation of the property statements of the RDF instances. Important to note is the seperation of the last two steps. As all RDF instances exists before the properties are created, it is possible to reference other RDF instances (from the same mapping or another).

An example of a mapping between a SQL query and RDF data definition is showed in listing 6.4

```
1  <d2r:ClassMap
2               type="test:doctor"
3               sql="SELECT␣*␣FROM␣doctor"
4               resourceIdPattern="@@id@@"
5               id="doctor">
6
7          <d2r:DataTypePropertyBridge property="doctor:id" pattern="
               @@id@@" dataType="xsd:integer"/>
8          <d2r:DataTypePropertyBridge property="doctor:firstname" pattern
               ="@@firstname@@" dataType="xsd:string"/>
9          <d2r:DataTypePropertyBridge property="doctor:lastname" pattern=
               "@@lastname@@" dataType="xsd:string"/>
10         <d2r:ObjectPropertyBridge property="doctor:hospital"
               referredClass="hospital" referredColumns="doctor.hospital"
               />
11
12         <d2r:MetaData>person doctor clinician</d2r:MetaData>
13     </d2r:ClassMap>
```

Listing 6.4: Example of a MeDSpace D2rMap mapping

In the listing relational data for doctors working in a hospital is mapped to RDF. The rdf type is specified in line **3**, and in line **4** the SQL query is specified that should be executed to collect the data.

The *DataTypePropertyBridge* tag defines rdf statements that have a literal value (e.g. the name of the doctor). With *ObjectPropertyBridge* rdf statements to other class mappings can be defined. In the above example the doctor class map references a hospital class map that is not shown in the listing.

Names that are enclosed by two '@@' specify a value of a column. E.g. with '@@firstname@@' is meant the value of the column *firstname* which is part of the sql query 'SELECT * FROM doctor'.

With the *MetaData* one can specify tags for the rdf data. This is useful for the keyword search in MeDSpace, as additional domain knowledge can be stated that should be searchable. That way it is possible to get doctor rdf data in the search result if you search for 'person' or 'clinician'.

6.3.3.1 MeDSpace reimplementation

Chris Bizer published in 2003 a processor for D2RMap. We used this implementation, but it was necessary to reimplement it, as the original implementation didn't meet our requirements. We did the following changes:

- The processor was changed to be used as a library.
- The processor was updated to Java 8.
- Type safety issues were resolved
- Jena dependency was removed. Library uses now RDF interfaces from MeDSpace. That allows to use the library with any RDF framework backend.
- XML configuration was adapted to the needs of MeDSpace.
- Manual XML configuration loading was replaced by JAXB.
- Previously an RDF graph was produced that was stored completely in memory. Now, RDF triples are created and streamed 'on demand'. Thus memory usage is capped. This also allows to handle very large RDF graphs.
- Previously the whole SQL result data was temporarily stored in memory. Now a fix fetch size is used. Thus memory usage is capped.

6.3.4 Keyword search

After discussing the SQL to RDF mapping we now look at the keyword search, now: Mainly there are two possibilities to implement a keyword search functionality:

- Construct a keyword search query in SQL and let the database answer the query
- Use a keyword search engine that answers the query based on an external index

At first glance the first option sounds obviously simple, but after implementing it showed several disadvantages: SQL is not designed to provide search functionality based on keywords. SQL uses the ***LIKE*** operator for pattern matching. But in order to do a Full-text search the SQL Query executor cannot use any index resulting in poor query answer performance. Another problem of *LIKE* is that there is no way to define searching only whole words and not just sub word matching. Whole word matching is very important, as e.g. a user searching for data about male patients should not get also data about female patients. As a result the *LIKE* operator is not suitable for a proper keyword search service as expected to be provided by a dataspace wrapper. Several SQL database vendors often provide own solutions for Full-Text search queries. But these solutions often have restrictions, as e.g. only column fields having the datatype *TEXT* (on MySQL), and the fields have to be specified as fulltext fields (in their creation or through an update operation) so that the SQL engine is able to create an index for it (at MySQL databases at least).
A Wrapper could use vendor specific services but obviously that would exclude other SQL database vendors.

The second option doesn't rise the aforementioned issues of option one. For the Wrapper a keyword searcher was implemented using the fulltext search engine Apache Lucene. The advantage of using Lucene is it's high-performance and scaling of keyword searches over large data sets. Additionally it allows a fine granular configuration about the query construction and sorts automatically the query result by relevance (so called query result ranking).
The major disadvantage of using lucene is that the SQL data have to be extracted

and indexed outside the database. If the data changes or rows are added resp. deleted, the index has to be updated accordingly. The update process can be very complex, as not only new data has to be indexed resp. existing data has to be removed, but also data that references the deleted, or new data that depends on it. A simpler but obviously slower solution is to reindex the whole data set. Reindexing the whole data set is only advised if updates occur not very often, or if acceptably, when the wrapper updates the index not instantly and thus provides potentially outdated data.

The decision which method is more suitable depends primarily on the use case and the domain. As the project is designed to be used as a test suite for medical datasources and medical science, it is acceptable if the data is outdated a bit to some degree and will be updated at every database change. So, changes on the datasource haven't to be updated in near-realtime.

Then one of the design goals for this project was to design a SQL wrapper that can be used with arbitrary vendors. As a result, vendor specific services are not an option.

All things considered the preferred method for the keyword search functionality clearly is Apache Lucene Core, as the advantages by far outweigh outweigh its disadvantages. Thus, a full functional keyword searcher was implemented powered by Lucene.

6.4 Image Wrapper

In this section details about the implementation of the image wrapper are presented. At first we want to look at the data the image wrapper is managing. Then we will discuss how the data is indexed, mapped, and converted to RDF. Finally we present the multimedia file download service that allows others to access the source files of DDSM cases.

6.4.1 DDSM

The data managed by the image wrapper are DDSM case file that are publicly available[18]. DDSM is a resource for mammographic image analysis and it contains data about approximately 2500 cases [40]. As one case needs relative much disk space, we decided to use only a subset of 50 cases that take altogether 3.7 GB.

In the following we will touch upon the content of a case, but for more details, please consult the specification of a DDSM case [47].

For each study a case contains two LJPEG images for each breast (so 4 images in total) and related patient information. LJPEG is a very old image format that nowadays isn't used very often anymore. In fact, we didn't found a tool, that could display it. Thus, we decided to convert the images to PNG. Hence, we used Sharma's *DDSMUtility* [48].

[18]`http://marathon.csee.usf.edu/Mammography/Database.html`

A case also contains a 16_PGM image that is an overview of the whole case. But the 16_PGM image is not really needed and as 16_PGM is a really old image format, we didn't used it.

Each case has a ICS file that holds general information about the case. It also specifies if a so-called overlay is defined for each breast image.

Overlay files are used to specify found abnormalities in the breast images. An overlay file contains the number of found abnormalities, the type of each abnormality, and defines outline boundaries which specify where the abnormality can be found in the source image.

6.4.2 RDF conversion

The data for each DDSM case is collected and useful data is extracted to create java objects from it. We created java classes for the ics file, overlay file, abnormality, and the lesion types, calcification and mass. The content of each java object, that is created using the aforementioned method, is indexed and stored locally.

When a keyword search query is executed, the searching module queries the indexed documents. Each document is assigned to a specific java class in order to know that a document represents an ICS file, an overlay file, etc. .

We defined for each type a RDF mapping that is specified in the configuration file *medspace-ddsm-mapping.xml*. The configuration file has a specification file *medspace-ddsm-mapping-specification.xsd*, so that the mapping types can be auto-generated using JAXB.

With the mapping configuration it is possible to generate RDF data from each document.

6.4.3 File downloading service

For the exported rdf types *IcsFile* and *Overlay*, which represent an ics file resp. overlay file of a DDSM case, we defined statements that point to a URL location of the source file. We decided to implement a service on the image wrapper that allows others to download these image files.

In listing 6.5 you can see an example of created RDF data for a breast image. The highlighted URL specifies the source image. In this example the image wrapper is running at localhost on port 9300, and as stated before, by following the link (e.g. in the browser) it would be possible to download the source image.

```
<http://www.medspace.com/images/ddsm/image#cancers/cancer_01/case0001/
    C_0001_1.LEFT_CC.png>
image:source "http://localhost:9300/get-file?relativePath=cancers/
    cancer_01/case0001/C_0001_1.LEFT_CC.png"^^xsd:anyURI .
```

Listing 6.5: image RDF conversion example

6.5 PDF Wrapper

In this section details about the implementation of the PDF wrapper are presented. At first we look at what data is managed by this wrapper. Then we take a closer look at how the data is indexed, mapped, and converted to RDF. Finally we present the multimedia file download service, that allows others to access the pdf files.

6.5.1 PDF Data

Actually it was planned to use existing PDF files, but we didn't found suitable PDF files that are publicly available and contain healthcare data. As a result, we generated our own PDF files. For this we used again Schmiedbauer's data set and wrote a little PDF generation program that establishes a JDBC connection to the MySQL database and fetches some data. From that data we created PDF files. For the PDF generation we used the framework Apache PDFBox [49].

6.5.2 RDF conversion

On startup the PDF wrapper extracts the content from the pdf files and indexes it. This way it is possible to do a full-text search over all pdf files.

The mapping to RDF is straight forward: Each pdf file is mapped to a rdf type called 'PdfFile'. The specification of this type is done in the file *medspace-pdf-wrapper-config-specification.xsd* which is used to create auto-generated java classes with JAXB. Also it is used to validate the mapping configuration file. The definition of the mapping is done in the corresponding configuration file *medspace-pdf-wrapper-config.xml*. Listing 6.6 shows an extract of the configuration file:

```
1 <pdf:PdfRootDirectory>./_work/pdfFiles/</pdf:PdfRootDirectory>
2 <pdf:PdfFile rdfType="http://www.medspace.com/pdf/pdf_file" classId="
      pdfFile">
3     <rdf-mapping:metaData>pdf file</rdf-mapping:metaData>
4     <pdf:source propertyType="pdf_file:source" dataType="xsd:string"/>
5 </pdf:PdfFile>
```

Listing 6.6: Extract of the pdf wrapper configuration

In line **1** the directory is specified, where all pdf files are located. Starting from line **2** is the definition of the RDF type *PdfFile* which contains the source file location and (optional) meta data tags (as the other wrappers use it).

6.5.3 File downloading service

This service works similar like the one of the image wrapper: every PDF file that is mapped to RDF is assigned a download link so that it can be downloaded from external programs.

Every pdf file gets assigned an ID so that the downloading service can distinguish and locate them. The ID thereby is the file path in relation to the pdf file root directory that is specified in the configuration file.

6.6 MeDSpace server

In this section implementation details of the MeDSpace server application are presented. The MeDSpace server combines all gloabal functionality of the MeDSpace system. This includes the modules *Query Executor*, *Data Collector*, and *Register*.

6.6.1 Starting the MeDSpace server

There are start scripts for Windows and Unix systems that can be found at *{medspace-root-folder}/medspace/bin* . It is recommended to use them so that all gets properly configured (library paths, etc.).

6.6.2 Query Caching

In this passage details about the implementation of the query caching are explained, a functionality that is implemented in the *Query Executor* module.

For each keyword search query we created unique strings in order to avoid caching duplicates. The algorithm works as follows: the keywords are transformed to lower case, trimmed, lexically sorted, and finally duplicates are deleted. For actually caching the query, we used EHCache [50]. It should be noted that only the queries itself are cached by EHCache. The result of the query is stored by the *Data Collector* using a *NativeStore*[19] from the RDF4J framework.

6.6.3 Datasource storing

In this section details about the implementation of the *Register* module are presented.

When the register is shutting down it saves all registered datasources into a local file *datasources.json* in the current working directory. The datasources are serialized to JSON in this file. If MeDSpace finds the aforementioned file, it will auto-register them on startup without contacting the respective datasources.

6.6.4 IO-Error policy

If a datasource doesn't respond anymore, e.g. the network is overloaded or the wrapper doesn't deregister itself for any reason, every try to call it's services would inevitable result in a lot of io-errors, slowing down query answering. Therefore we implemented an io-error policy that works as follows: for every registered datasource a counter for the happened io-errors is hold. Whenever an io-error occurs the counter gets increased. Should the counter exceed a specified limit, the datasource gets deregistered automatically. The default value of the io-error limit is 5, but this can be configured in the *medspace-config.xml* from the *medspace* project.

[19] `http://docs.rdf4j.org/javadoc/2.2/org/eclipse/rdf4j/sail/nativerdf/NativeStore.html`

6.7 GUI

In this section is explained how the GUI of MeDSpace was created.

For dynamic (html) page creation we used Play's template engine *Twirl* [20]. The templates are normal text files that are enriched by Scala code [51].

For styling the html content, we used SASS, an extension of CSS that is used to generate CSS files [52]. In order to auto-compile the SASS files we've used the *sbt-sassify* plugin [21]. The SASS source file which we used for MeDSpace can be found at *{medspace-root-folder}/project-sources/medspace/app/assets/stylesheets/*

The look of the search page is illustrated in figure 6.3:

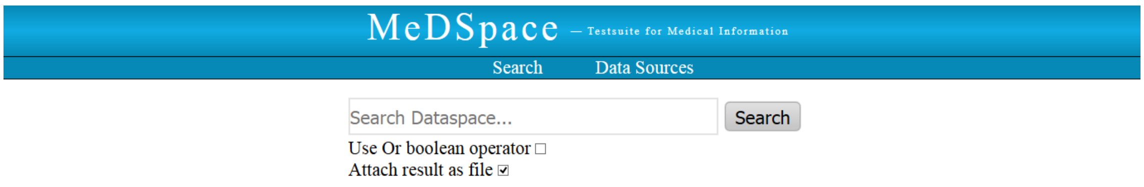

MeDSpace – Testsuite for Medical Information
Search
Data Sources
Search Dataspace...
Search
Use Or boolean operator ☐
Attach result as file ☑

Figure 6.3: MeDSpace search page

The search page contains a blue header with the writing 'MeDSpace - Testsuite for Medical Information'. Then the navigation menu follows which contains buttons to navigate to the search and datasources page. For each button exists a tooltip describing the respective page. An example of this tooltips is illustrated in figure 6.4.

In the page body is an input field allowing a user to state a keyword search query. The user can state if he wishes to use the OR operator instead of the AND operator, and he can state whether he wants the rdf query result to be returned as a file. Both options can be activated by using the check boxes under the text input field. With the *Search* button the query gets executed.

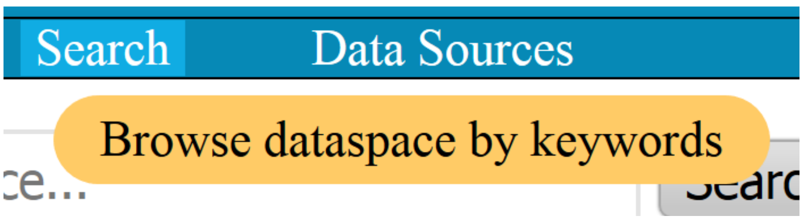
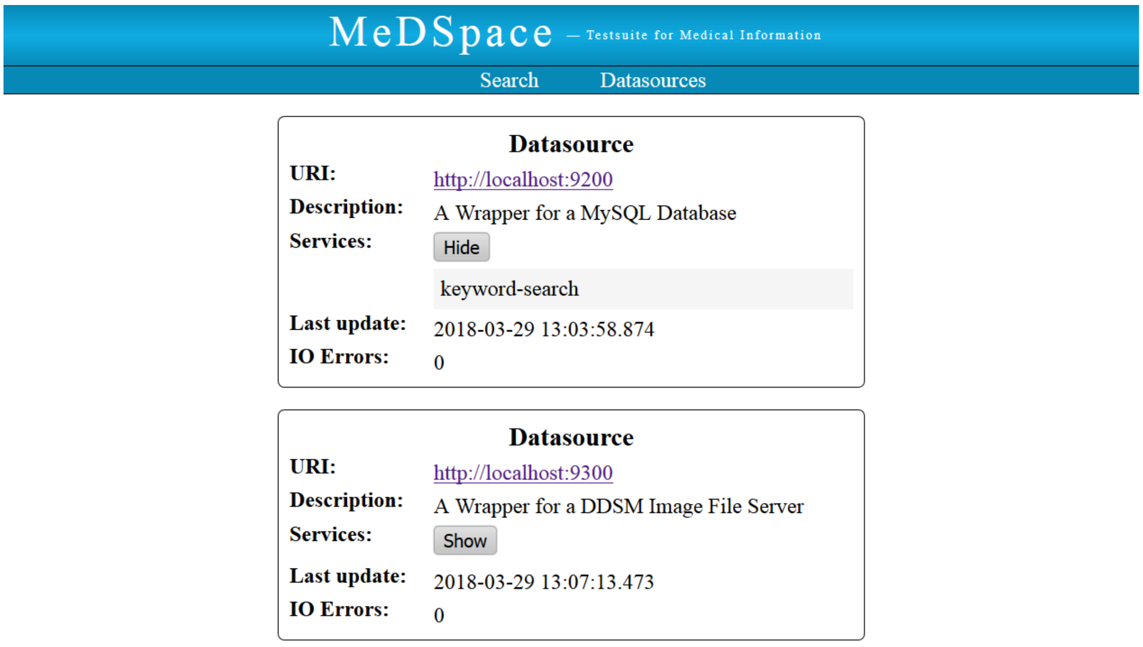


Figure 6.4: Tooltip of a navigation button

The *Data Sources* page lists all registered datasources, as illustrated in figure 6.5:

[20]https://github.com/playframework/twirl
[21]https://github.com/irundaia/sbt-sassify



Datasource

URI:	http://localhost:9200
Description:	A Wrapper for a MySQL Database
Services:	Hide
	keyword-search
Last update:	2018-03-29 13:03:58.874
IO Errors:	0

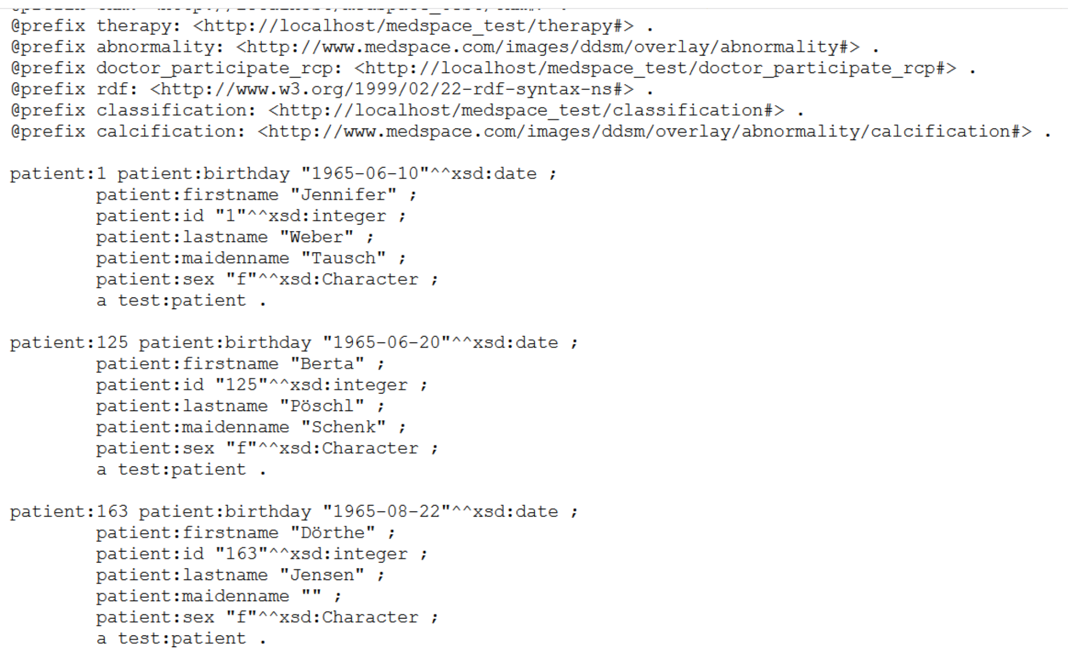
Datasource

URI:	http://localhost:9300
Description:	A Wrapper for a DDSM Image File Server
Services:	Show
Last update:	2018-03-29 13:07:13.473
IO Errors:	0

Figure 6.5: Data Sources page

For each datasource is listed its URL, a description of it, what services it provides, when it registered itself, and how much io errors occurred since its registration. The services are initially hidden and can be made visible by clicking the 'Show' button.

In figure 6.6 an extract of a query result is shown, that is displayed in the browser. The used keyword search query was 'patient 1965'. Thus it was searched for patients that also have a relation to a number '1965' which for example could be the year of birth.



```
@prefix therapy: <http://localhost/medspace_test/therapy#> .
@prefix abnormality: <http://www.medspace.com/images/ddsm/overlay/abnormality#> .
@prefix doctor_participate_rcp: <http://localhost/medspace_test/doctor_participate_rcp#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix classification: <http://localhost/medspace_test/classification#> .
@prefix calcification: <http://www.medspace.com/images/ddsm/overlay/abnormality/calcification#> .

patient:1 patient:birthday "1965-06-10"^^xsd:date ;
        patient:firstname "Jennifer" ;
        patient:id "1"^^xsd:integer ;
        patient:lastname "Weber" ;
        patient:maidenname "Tausch" ;
        patient:sex "f"^^xsd:Character ;
        a test:patient .

patient:125 patient:birthday "1965-06-20"^^xsd:date ;
        patient:firstname "Berta" ;
        patient:id "125"^^xsd:integer ;
        patient:lastname "Pöschl" ;
        patient:maidenname "Schenk" ;
        patient:sex "f"^^xsd:Character ;
        a test:patient .

patient:163 patient:birthday "1965-08-22"^^xsd:date ;
        patient:firstname "Dörthe" ;
        patient:id "163"^^xsd:integer ;
        patient:lastname "Jensen" ;
        patient:maidenname "" ;
        patient:sex "f"^^xsd:Character ;
        a test:patient .
```

Figure 6.6: Extract of a query result

7 Conclusion

In this thesis we presented MeDSpace, a testsuite for medical data. It allows to execute a distributed keyword search on heterogeneous datasources, containing medical information. Additionally, this system provides wrappers for a relational datasource that uses SQL for querying, a DDSM image file server and a PDF file server. The wrappers are able to register and deregister themselves from the MeDSpace system by calling the corresponding register and deregister REST services. MeDSpace supports query and search result caching, keyword search using AND, or optional OR operators, and query results can be displayed directly in the browser or downloaded as files. Additionally the system has an io-error policy for datasources, so that dead datasources are removed automatically. Search results of images and pdf files contain URLs to the source files. The wrappers for the respective datasource provide a service so that the source files can be downloaded.

The system was designed to be used as a test environment for implementing dataspace features. It can also be used as a starting point for a much more evolved dataspace.

There are many fields that could be improved: the system allows to cache queries. This could be used as a starting point for implementing incremental (*pay-as-you-go*) data integration which would move MeDSpace more into the direction to be an actual dataspace. Furthermore, MeDSpace uses RDF as its canonical data model. Therefore an ontology-based data integration service could be implemented that uses ontologies specified in OWL. Currently MeDSpace also uses a very primitive method to do the keyword search query. A possible improvement here would be to implement a global query language that considers also multimedia data. A starting point could be this work [53].

Bibliography

[1] U. Leser and F. Naumann, *Informationsintegration: Architekturen und Methoden zur Integration verteilter und heterogener Datenquellen.* dpunkt, 2006.

[2] I. Elsayed, P. Brezany, and A. Tjoa, "Towards realization of dataspaces," in *Database and Expert Systems Applications, 2006. DEXA '06. 17th International Workshop on*, pp. 266–272, 2006.

[3] A. D. Sarma, X. L. Dong, and A. Y. Halevy, "Data modeling in dataspace support platforms," in *Conceptual Modeling: Foundations and Applications - Essays in Honor of John Mylopoulos*, pp. 122–138, 2009.

[4] S. H. R. Wurst, *Dataspace Integration in der medizinischen Forschung.* PhD thesis, Technische Universitaet Muenchen, 2010.

[5] C. Bizer, "D2r map – a database to rdf mapping language," 01 2003.

[6] D. Tomar, "A survey on data mining approaches for healthcare," vol. 5, pp. 241–266, 10 2013.

[7] C. Yu and T. Brandenburg, "Multimedia database applications: Issues and concerns for classroom teaching," *CoRR*, vol. abs/1102.5769, 2011.

[8] W. Raghupathi and V. Raghupathi, "Big data analytics in healthcare: promise and potential," *Health Information Science and Systems*, vol. 2, p. 3, Feb 2014.

[9] M. Franklin, A. Halevy, and D. Maier, "From databases to dataspaces: A new abstraction for information management," *SIGMOD Rec.*, vol. 34, pp. 27–33, Dec. 2005.

[10] A. Ndjafa, H. Kosch, D. Coquil, and L. Brunie, "Towards a model for multimedia dataspaces," in *Multimedia on the Web (MMWeb), 2011 Workshop on*, pp. 33–37, Sept 2011.

[11] M. Lenzerini, "Data integration: A theoretical perspective," in *Proceedings of the Twenty-first ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '02, (New York, NY, USA), pp. 233–246, ACM, 2002.

[12] I. Limited, *Introduction to Database Systems.* Pearson Education, 2010.

[13] A. P. Sheth and J. A. Larson, "Federated database systems for managing distributed, heterogeneous, and autonomous databases," *ACM Comput. Surv.*, vol. 22, pp. 183–236, Sept. 1990.

[14] H. T. C. M. Sperberg-McQueen, "Xml schema." `https://www.w3.org/XML/Schema`, 2000. Last accessed on 02/15/2018.

[15] D. Tsichritzis and A. Klug, "The ansi/x3/sparc dbms framework report of the study group on database management systems," *Information Systems*, vol. 3, no. 3, pp. 173 – 191, 1978.

[16] M. T. Roth and P. M. Schwarz, "Don't scrap it, wrap it! a wrapper architecture for legacy data sources," in *Proceedings of the 23rd International Conference on Very Large Data Bases*, VLDB '97, (San Francisco, CA, USA), pp. 266–275, Morgan Kaufmann Publishers Inc., 1997.

[17] G. Wiederhold, M. Genesereth, and M. Papazoglou, "The conceptual basis for mediation services mail to issue editor the conceptual basis for mediation services," 1996.

[18] M. T. O. Ling Liu, ed., *Encyclopedia of Database Systems.* Springer, 1 ed., 2009.

[19] S. Vatika and D. Meenu, "Sql and nosql databases," *International Journal of Advanced Research in Computer Science and Software Engineering*, 2012.

[20] A. Halevy, M. Franklin, and D. Maier, "Principles of dataspace systems," in *Proceedings of the Twenty-fifth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '06, (New York, NY, USA), pp. 1–9, ACM, 2006.

[21] S. Agrawal, S. Chaudhuri, and G. Das, "Dbxplorer: a system for keyword-based search over relational databases," in *Data Engineering, 2002. Proceedings. 18th International Conference on*, pp. 5–16, 2002.

[22] L. Guo, F. Shao, C. Botev, and J. Shanmugasundaram, "Xrank: Ranked keyword search over xml documents," in *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, SIGMOD '03, (New York, NY, USA), pp. 16–27, ACM, 2003.

[23] V. Hristidis and Y. Papakonstantinou, "Discover: Keyword search in relational databases," in *Proceedings of the 28th International Conference on Very Large Data Bases*, VLDB '02, pp. 670–681, VLDB Endowment, 2002.

[24] M. Carey, D. Florescu, Z. Ives, Y. Lu, J. Shanmugasundaram, E. Shekita, and S. Subramanian, "Xperanto: Publishing object-relational data as xml," in *In WebDB*, pp. 105–110, 2000.

[25] R. Krishnamurthy, V. Chakaravarthy, R. Kaushik, and J. Naughton, "Recursive xml schemas, recursive xml queries, and relational storage: Xml-to-sql query translation," in *Data Engineering, 2004. Proceedings. 20th International Conference on*, pp. 42–53, March 2004.

[26] Q. He, L. Qiu, G. Zhao, and S. Wang, "Text categorization based on domain ontology.," in *WISE* (X. Zhou, S. Y. W. Su, M. P. Papazoglou, M. E. Orlowska, and K. G. Jeffery, eds.), vol. 3306 of *Lecture Notes in Computer Science*, pp. 319–324, Springer, 2004.

[27] H. Zerrik, "Repräsentation eines dataspace-modells in rdf," 2013. Unpublished bachelor thesis, that was submitted to the university of passau, chair of distributed systems.

[28] "Debugit: Detecting and eliminating bacteria using information technology. a large-scale integrating project." online article.

[29] D. Schober, R. Choquet, K. Depraetere, F. Enders, P. Daumke, M. Jaulent, D. Teodoro, E. Pasche, C. Lovis, and M. Boeker, "Debugit: Ontology-mediated

layered data integration for real-time antibiotics resistance surveillance," in *Proceedings of the 7th International Workshop on Semantic Web Applications and Tools for Life Sciences, Berlin, Germany, December 9-11, 2014.*, 2014.

[30] P. Daumke, *Das MorphoSaurus-System - Lösungen für die linguistischen Herausforderungen des Information Retrievals in der Medizin.* PhD thesis, Universitaet Freiburg, 2007.

[31] R. C. D. T. F. E. C. D. M.-C. J. Ariane Assélé Kama, Audi Primadhanty, "Data definition ontology for clinical data integration and querying," *Studies in Health Technology and Informatics*, vol. Volume 180: Quality of Life through Quality of Information, pp. 38 – 42, IOS Press 2012.

[32] J. Carter, *Electronic Health Records: A Guide for Clinicians and Administrators.* American College of Physicians, 2008.

[33] D. Schober, I. Tudose, and M. Boeker, "Developing dco: The debugit core ontology for antibiotics resistence modelling."

[34] "Rdf." `https://www.w3.org/RDF/`, 2014. Last accessed on 02/10/2018.

[35] T. Berners-Lee, "Linked data." `https://www.w3.org/DesignIssues/LinkedData.html`, 2006. Last accessed on 03/22/2018.

[36] B. M. Ed Ort, "Java architecture for xml binding (jaxb)." `http://www.oracle.com/technetwork/articles/javase/index-140168.html`, March 2003. Last accessed on 02/10/2018.

[37] "About the unicode® standard." `http://unicode.org/standard/standard.html`. Accessed: 2017-04-07.

[38] "Html5 a vocabulary and associated apis for html and xhtml w3c recommendation 28 october 2014." `https://www.w3.org/TR/html5/document-metadata.html#charset`. Accessed: 2017-04-07.

[39] M. Schmidbauer, "Medical data generator." Unpublished Bachelor thesis; Accessible through the libary of the university of passau, 2012.

[40] "University of south florida digital mammography home page." `http://marathon.csee.usf.edu/Mammography/Database.html`. Last accessed on 02/10/2018.

[41] G. L. Taboada, S. Ramos, R. R. Expósito, J. Touriño, and R. Doallo, "Java in the high performance computing arena: Research, practice and experience," *Science of Computer Programming*, vol. 78, no. 5, pp. 425 – 444, 2013. Special section: Principles and Practice of Programming in Java 2009/2010 & Special section: Self-Organizing Coordination.

[42] "The eclipse rdf4j framework." `http://rdf4j.org/`, 2018. Last accessed on 02/10/2018.

[43] "Apache jena." `https://jena.apache.org/`, 2011. Last accessed on 03/22/2018.

[44] "Apache lucene core." `https://lucene.apache.org/core/`, 2016. Last accessed on 02/10/2018.

[45] "Play framework." `https://www.playframework.com/`. Last accessed on 02/10/2018.

[46] "Sbt - the interactive build tool." `https://www.scala-sbt.org/`. Last accessed on 03/24/2018.

[47] "Ddsm: Description of a case." `http://marathon.csee.usf.edu/Mammography/DDSM/case_description.html`. Last accessed on 03/22/2018.

[48] A. Sharma, "Ddsm utility github project." `https://github.com/trane293/DDSMUtility`, 2016. Last accessed on 02/10/2018.

[49] "Apache pdfbox." `https://pdfbox.apache.org/`, 2018. Last accessed on 02/10/2018.

[50] "Ehcache - java's most widely-used cache." `http://www.ehcache.org/`. Last accessed on 03/26/2018.

[51] "The template engine." `https://www.playframework.com/documentation/2.6.x/JavaTemplates`. Last accessed on 03/26/2018.

[52] "Sass." `https://sass-lang.com/`. Last accessed on 03/26/2018.

[53] M. Doller, A. N. N. Yakou, R. Tous, J. Delgado, M. Gruhne, M. Choi, and T. B. Lim, "Semantic mpeg query format validation and processing," *IEEE MultiMedia*, vol. 16, pp. 22–33, April 2009.

Eidesstattliche Erklärung

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Die Arbeit wurde bisher keiner anderen Prüfungsbehörde vorgelegt und auch noch nicht veröffentlicht.

Passau, den 29.03.2018

David Goeth