VRIJE
UNIVERSITEIT
BRUSSEL

OPEN INFORMATION SYSTEMS

Milestone 4: Using Ontologies and Final Report

Group 4:

0520238 Robrecht Blancquaert
robrecht.simon.blancquaert@vub.be
0568138 Denis Carnier
denis.carnier@vub.be
0564574 Gwij Grenié
gwij.maria.d.grenie@vub.be
0565683 Dieter Vandesande
dieter.vandesande@vub.be

December 27, 2020

Science and Bio-Engineering Sciences

Abstract

Before 2020, blended learning at educational institutes wasn't very common, with full online courses even being the exception rather than the rule. But as of December 2020, the necessity for online learning has shot up because of the COVID-19 pandemic, with many online lectures and video conferences being held concurrently. As a result, a need to represent the information they hold has risen. Therefore, we propose the creation of an open information system to help tackle this problem. The aim of this report is to describe the development and results of such a system, highlighting the design choices we made, while also providing ways to implement, validate and use the proposed system. First, we provide the conceptual schema which captures the information about online learning and video conferencing. Based on this schema is the heart of the system, an ontology. While already useful on its own, we also describe ways to use this ontology through a proof-of-concept web application, to map SQL data to RDF data through the use of R2RML mappings and to validate information before addition through the use of SHACL.

1 Introduction

The primary purpose of this report is to describe all phases which we went through to develop and test the needed information system. We both motivate and discuss its design, but also provide potential uses of the system through a demonstrator and a simple proof-of-concept. We also pay special attention to validation of information to be added to the system.

First, in section 2, we summarise the division of the workload, in essence which team member handled which task. As this project is rather on the large side, we had to make a good and fair division so we may accomplish this project on time, while also trying to maintain a healthy workload.

Next, in section 3, we showcase the conceptual schema created using ERD and provide a motivation of its design. The particular subject, which our system handles, contains many freedoms, which may pose a challenge to model. A teacher may for example choose to use Panopto to record his class, while also allowing a small percentage of students to take the class face-to-face. Another teacher may choose to completely teach his course online, using Microsoft Teams, or some other conferencing service. These freedoms form an aspect which our system must be able to handle, and thus need to be captured by our conceptual model.

We proceed in section 4 to transform this conceptual model into a solid ontology and explain the design choices we made for the ontology. Some important axioms of the ontology get special care here.

Afterwards, in section 5 we discuss which changes and extensions our ontology underwent to fix any issues we later experienced in the project. Next in this same section, we give an outline of R2RML, where we map data from an SQLite database, to RDF triples to be used with our ontology. Third in this section, we discuss how to validate information to be added to the graph. As RDF cannot provide any guarantees that no absurd, incomplete or inconsistent information can be added, there is a need for our system to first validate data before addition. Last in this section, we discuss setting up a linked data frontend for our ontology.

Section 6 provides a simple proof of concept that uses our ontology to showcase its usefulness.

The last section, section 7, provides the needed information to reproduce the results and to access the created artifacts.

Finally, in section 8 we give a concise conclusion.

2 Division of workload

During the project we strived to create a balanced workload for each member by creating a list of tasks and dividing the work equally. We focused on helping each other with the tasks and working together to overcome obstacles to achieve a project we could all be proud of.

Milestone 1: Conceptual schema

In the first phase we focused on brainstorming about the ontology we wanted to create and the implications of our ideas for further stages of the project. The deliverables, namely the ERD and the report, were created by Dieter and Denis.

Milestone 2: Ontologies

Next, we focused on implementing an ontology of the ideas described during the first phase, and on expanding and adjusting the scope of what we wanted to represent in the ontology. For this phase, Gwij and Robrecht created an ontology file and wrote the report.

Milestone 3: Intermediate presentations

The intermediate presentations were a collaboration of all four members. Everyone focused on one quarter of the presentation.

Milestone 4: Final Ontologies

For the final phase we largely all worked together in meetings to finalise the project, but the division of tasks can be broadly sketched as follows: Denis created frontend and

environment setup, Dieter made the R2RML mappings and database, Robrecht for SQL and SPARQL queries and implementing them in the mappings, and Gwij implemented the SHACL validation and edited the ontology as needed.

The final report was made by all four members in equal parts.

3 Conceptual schema

We begin by highlighting several decisions we made in our schema. The full schema can be found in Appendix A.

Course Students can take courses. A course is uniquely identified by the course number, which is created by the university.

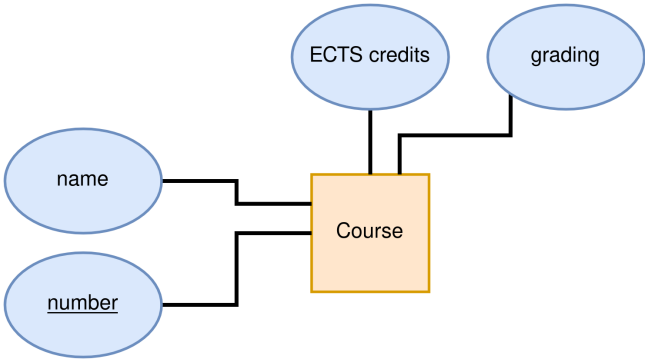

Figure 1: A Course entity and its attributes

Persons As shown in figure 2, the entity `Person` has two subclasses, namely `Teacher` and `Student`. A person has to be a teacher or a student, but they can be both at the same time. For this reason, we placed a specialisation constraint but we did not place a disjoint constraint. A teacher can be a professor, a teaching assistant or a guest lecturer, and is uniquely identified by their username. It is not possible for a teacher to be both a teaching assistant and a professor. Therefore, we have chosen a disjoint constraint. We have also placed a total specialisation constraint, which makes sure that a teacher is always either a professor or a teaching assistant.

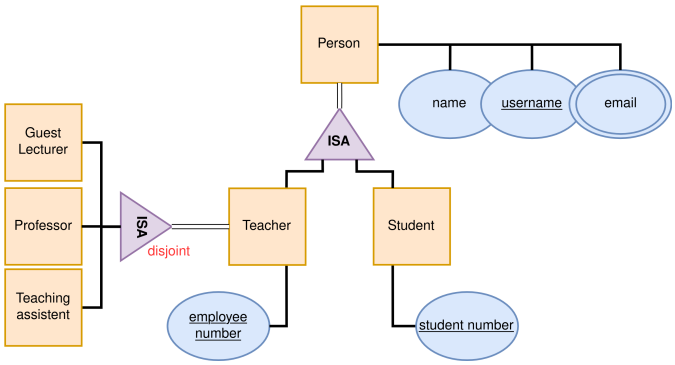

Figure 2: The Person entity, its subclasses and its attributes

Class The class entity is a representation of a *class* that is taught at a certain moment in time. A class can be a *lab session*, a *lecture* or a *Q&A session*. Both a total specialisation constraint and a disjoint constraint are used, in order to be sure that a *class* is always exactly one of the three aforementioned types.

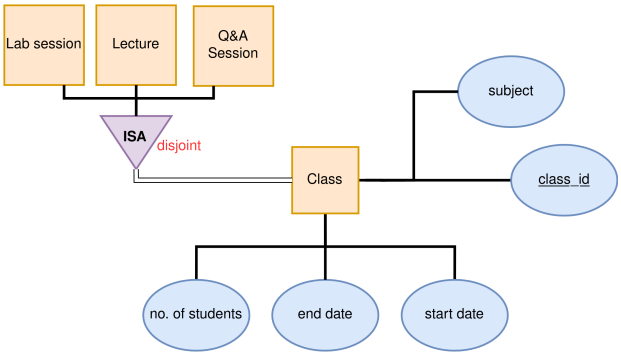

Figure 3: A class entity and its attributes

At first, we made the entity class a weak entity and used the teacher as part of the primary key together with the start date of the class. We received the feedback that the teacher can be unknown in advance or that the teacher can change. Therefore, we decided to use an auto-generated primary key for the class entity.

Course Material During the classes, a teacher can use course material. This can be one of the following types: a textbook, presentation handouts, example code, exercise sheets or lecture notes. Both a total specialisation constraint and a disjoint constraint are used here, such that the type of the course material has to be clear.

Because a class, its course, the teacher and the course material is relied on eachother, figure 4 shows how they are linked to eachother.

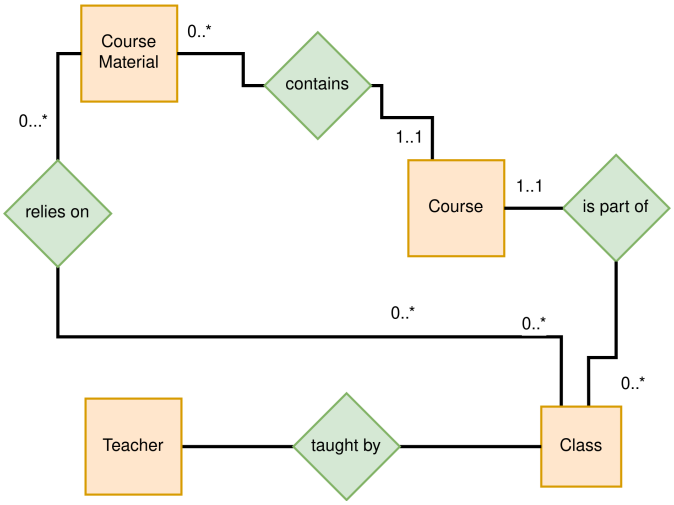

Figure 4: Relationship between main entities

Classroom A classroom has to represent either a *physical classroom* or an *online classroom*. Therefore, both a disjoint constraint and a specialisation constraint are placed on this ISA relationship.

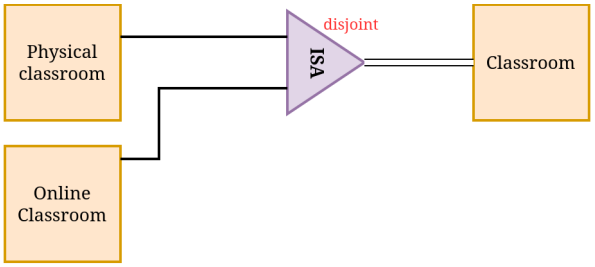

Figure 5: ISA relationship of a classroom entity

In Figure 6 the *physical classroom* entity is shown. It is uniquely identified by the physical room number.

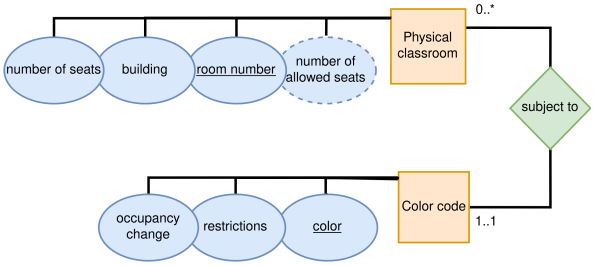

Figure 6: A Physical classroom entity and its attributes

Furthermore, a Microsoft Teams meeting, a Zoom meeting, a Big Blue Button conference or a lesson in a physical classroom can be recorded. To have a better support to save these recordings, we created the recording entity, which is shown in figure 12

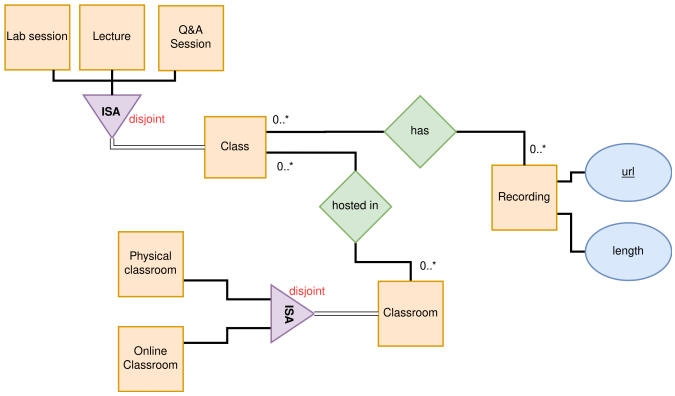

Figure 7: Recording of classroom

Following the feedback after the first iteration, physical classrooms have been extended to allow for COVID-specific occupancy, aptly named number of allowed seats in our model. Figure 3 shows that the attribute is a computed result from the accompanying active Color code entity instance associated with that particular classroom, and the maximum occupancy under normal conditions, stored in an attribute named number of seats. The Color code entity has an associated color attribute that single handedly defines its uniqueness, and restrictions that apply to that color code.
Figure 10 shows the entities for a Zoom meeting, a Panopto Recording and a Microsoft Teams meeting. A Panopto Recording can have annotations and bookmarks attached to it, which are created by a Person. Figure 9 shows the entity of a BigBlueButton conference. A breakout room can be created within a BigBlueButton conference and multiple people can be assigned to such a breakout room.

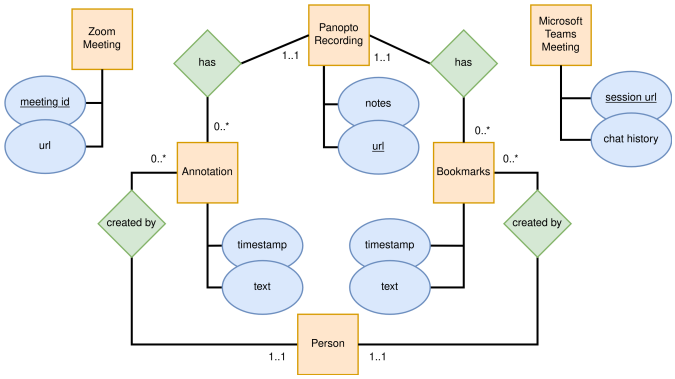

Figure 8: Zoom - Panopto - Teams entities

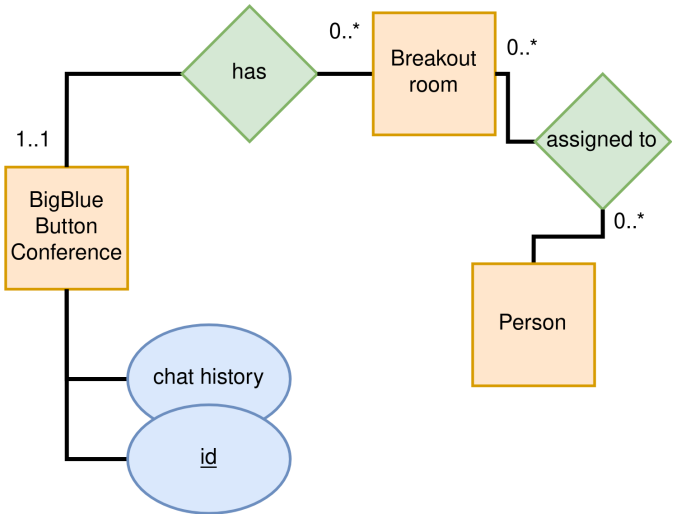

Figure 9: Big Blue Button entities

We made a few changes to this concept. First of all, we extended the possibility to store recordings of online meetings. Therefore, we created a new Concept named Recording. In the second iteration, there was still a blob attached to a Recording. In the final ERD, we use an url to point to the location of a Recording.

Figure ?? shows that within a BigBlueButton conference, a poll can be created with a question. Students can vote for poll options, which are attached to one specific poll.

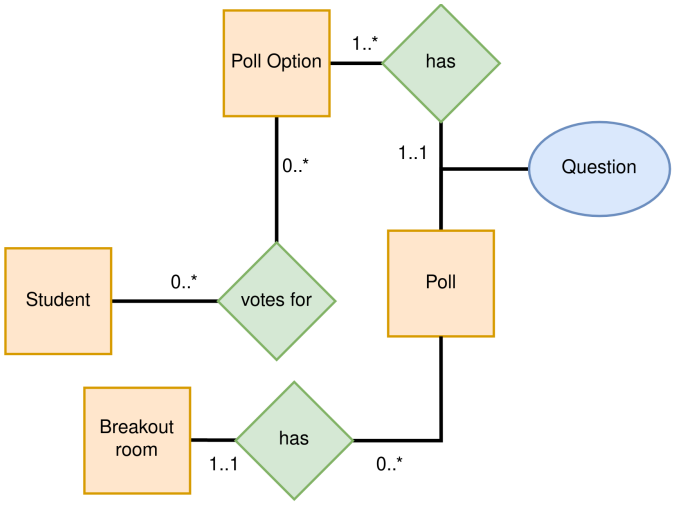

Figure 10: Poll entities

4 Ontology

This section was borrowed in large parts from our second milestone report titled *Milestone 2: Ontology*. It is included for completeness' sake. Some of the design decisions regarding the ontology showcased in this section might be subject to changes outlined in 5: Final ontology.

4.1 Main components of the ontology

In this section, we start by presenting the main components of our ontology. After that, we discuss how our ontology is built up in OWL[7], the Web Ontology Language.

4.1.1 Courses and people

A course can be seen as the central element of the ontology. It has several datatype properties such as a course code and the amount of credits it requires. More interesting are its object property relationships. A course namely has students, teachers, course materials, and of course classes. Students and teachers are both subclasses of the external object Person, which is retrieved from the FOAF ontology[2]. As expected, the Person object contains most of the values associated with people, and our specific ontology only expands on Teacher and Student with the natural properties they would have within the context of the university. These properties are the student and employee id given out by the university, and what types a teacher can be represented by object properties to Professor, Teaching assistant, and Guest lecturer.
The course material object is mainly used to refer to an external Work object from DBpedia[1], while containing some extra datatype properties to keep track of when the material was added or modified. Any course material can also be used in a specific class.

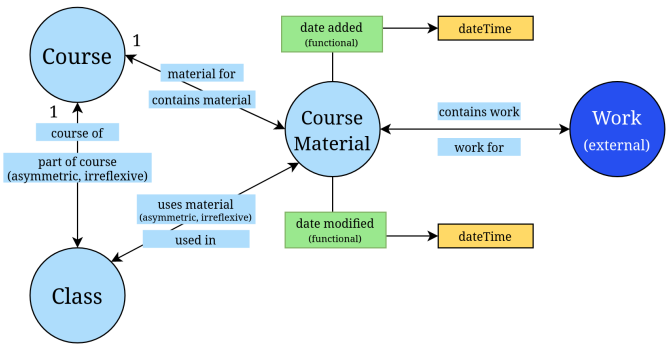

Figure 11: Course and course material

4.1.2 Classes and classrooms

A class also contains relationships to Student and Teacher like course, as the students and teachers per class can differ from those of the Course for which a class is given. Additionally a class contains information about when it is given and a specific subject.
The main property of Class within our ontology is that, in these corona times, classes are not only given in physical classrooms, but also in online environments. For this we have modelled a Classroom object that has no specific properties, except that it can be one of the aforementioned types. Separate from this is the fact that a class can have a recording, as both physical and online classes can be recorded.

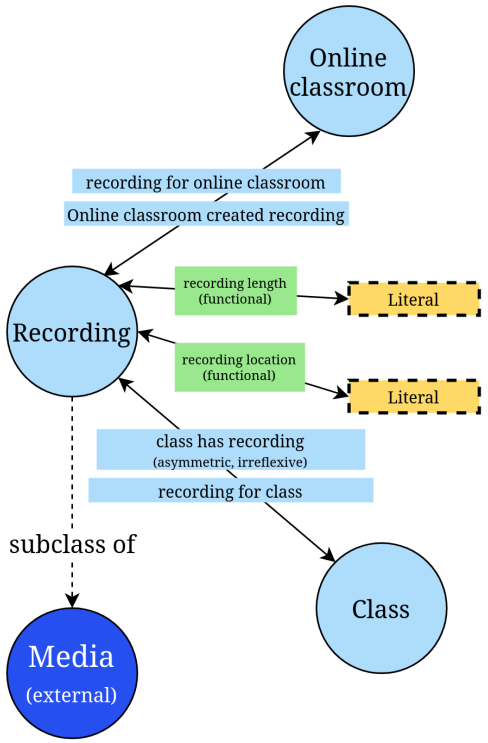

Figure 12: Recording relationships

For physical classrooms, in addition to the standard properties such as location and number of seats, we have an object property that assigns a Color code. This contains a value describing the color, such as *red*, and a description of the restrictions. Interesting is that the occupancy change is also recorded, in such a way that it is used together with the number of seats in a physical classroom to determine how many people are allowed in this particular classroom.
For online classrooms we differentiate between the two classes Online meeting and panopto recording. The difference between these two is that on the one hand online meetings are live and have a chat history. The panopto recordings on the other hand have a few different object relations to classes such as Bookmarks and Annotations, which can be added by a specific Teacher or Student respectively.
Within online meetings, there are also several different subclasses. Two of these are Microsoft Teams meetings with a session url and Zoom meetings with a meeting id. More interesting is the Big Blue Button meeting with several interesting object properties. One of these is a Poll, with Poll options that can be voted on by students. Additionally, there are also Breakout rooms that can be attended by the union of Teacher and Students. We chose to use a union here instead of just the Person class to get more granularity within the ontology.

4.2 Representation in OWL

We outline how the ontology is represented in OWL through examples.
Listing 1 gives the definition of the owl class PhysicalClassroom.

```
:PhysicalClassroom
  a owl:Class ;
```

```
  rdfs:label "Physical classroom"@en ;
  rdfs:comment "A physical classroom where
  ↪  classes are taught in."@en ;
  rdfs:subClassOf :Classroom ;
  owl:equivalentClass [
    a owl:Restriction ;
    owl:onProperty :subjectToColorCode ;
    owl:cardinality 1 ] .
```

Listing 1: OWL declaration of PhysicalClassroom

Note the restriction on the property `:subjectToColorCode`. This means that a `PhysicalClassroom` always needs to be subject to exactly one color code. Furthermore, a `PhysicalClassroom` is a subclass of the `Classroom` class and is defined as such.

Next, we define the object properties. These can be seen as the relationships between individuals of the classes defined above. In listing 2 we discuss the implementation of the object property `classroomCreatedRecording` and its inverse property `recordingForClassroom`. First of all, the domain of the property is the union of the classes `OnlineMeeting` and `PhysicalClassroom`. This means that both an online meeting and a physical classroom can have a recording. The inverse property is also defined, allowing a reasoner to infer the relationship the other way around.
Furthermore, an object property can have some characteristics as well. In this case, the property is asymmetric, which means that the property is not valid when the subject and the object are inversed, we have to use the inverse property for that. The property "classroomCreatedRecording" is also irreflexive, which means that a recording cannot have created itself and that a classroom cannot be a recording created by itself.

```
:classroomCreatedRecording
  a owl:ObjectProperty ,
    owl:AsymmetricProperty ,
    owl:IrreflexiveProperty ;
  rdfs:label "Online classroom created
  ↪  recording"@en ;
  rdfs:comment "Assigns an online classroom to
  ↪  a recording. Inverse property:
  ↪  recordingForClassroom"@en ;
  rdfs:domain [ owl:unionOf (:OnlineMeeting
  ↪  :PhysicalClassroom)] ;
  rdfs:range :Recording .

:recordingForClassroom
  a owl:ObjectProperty ;
  rdfs:label "recording for online
  ↪  classroom"@en ;
  rdfs:comment "Assigns a recording to a online
  ↪  classroom"@en ;
  owl:inverseOf :classroomCreatedRecording .
```

Listing 2: OWL declaration of classroomCreatedRecording and its inverse property recordingForClassroom

After the definitions of the object properties, the datatype properties are defined. These datatype properties are the relations between the classes and the literals. An example for the `employeeId` of a teacher is shown in listing 3. Important to note is that both datatype properties and object properties can be used as subclasses of the domain.

```
:employeeId
  a owl:DatatypeProperty ,
    owl:FunctionalProperty ;
  rdfs:label "employee id"@en ;
  rdfs:comment "The identifier of an
  ↪  employee used by the university."@en
  ↪  ;
  rdfs:domain :Teacher ;
  rdfs:range xsd:string .
```

Listing 3: OWL declaration of employeeId property

Lastly, some general axioms are defined. Listing 4 shows such an example. The axiom defined tells us that an individual cannot be of more than one type of the listed ones, namely Professor, TeachingAssistant and GuestLecturer.

```
[ a owl:AllDisjointClasses ;
  owl:members (
    :Professor
    :TeachingAssistant
    :GuestLecturer ) ] .
```

Listing 4: OWL axiom definition

5 Final ontology

This section explores the modifications we made to the initial ontology by realizing the R2RML mappings, SHACL validation and set up of the linked data frontend.

5.1 Design changes and extensions

The final design of the ontology does not deviate too much from the original design. We did make some significant changes. The first part of this section gives an overview of the significant changes we made. Apart from these main changes, some more changes we made to add missing data properties (course name, poll question text ...) and other minor fixes such as wrong data types or typos in names, such as *assistent* to *assistant*. We also removed our property `:username` and instead moved over to using `foaf:nick`, as this is semantically similar enough to justify its use.

5.1.1 oneOf property

A first change we made was to remove the `owl:oneOf` predicate from some classes. From the description of this predicate in the owl specification, we got the wrong view on the actual meaning of this predicate. In summary, we thought that this allowed to specify that a class can only contain instances of some subclasses. But in reality, this predicate specifies that a class can only contain certain individuals. An example may illustrate this. Suppose we have the class of solar-system planets. In the original meaning we thought it allowed to specify that this class can only

exist of instances of the subclasses gas giants, terrestrial planets, ice giants and dwarf planets. But in the actual meaning, this predicate allows to specify that this class can only exist of the individuals Mars, Earth, Venus, etc. This caused many inference problems and even contradictions. We eventually removed this predicate where it was not appropriate.

5.1.2 CourseMaterial subclasses

A second change we made was the addition of four subclasses of `:CourseMaterial`, being `:CodeExample`, `:Book`, `:Handouts` and `:ExerciseSheet`. These are present in the ERD, as seen in figure 13 but we unfortunately forgot to model these in our ontology.

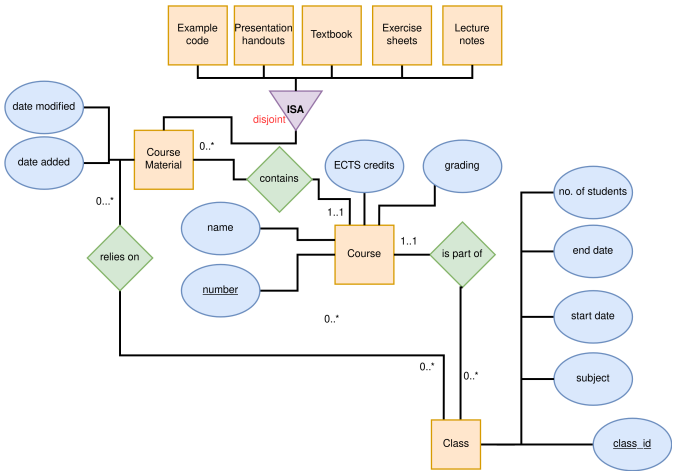

Figure 13: CourseMaterial entity and its relationships

5.1.3 Work external class

A third significant change we made was to move from `dbpedia:Work` to `dbo:file`. The reason for this is mainly technical. Still unknown to us, our reasoner crashed while trying to reason about `dbpedia:Work`. To not waste too many time trying to fix this issue, we moved to a different external class, which is semantically similar. Our search lead us to `dbo:file`, which did not give any technical problems.

5.1.4 Union domains

Another significant change we made was to change the domain of some object properties from the intersection of classes to the union of classes. This was an oversight from us which led to odd reasoning. Take for example the object property `:assignedToBreakoutRoom`. This property has range `:BreakoutRoom` and, in our original ontology, domain `intersection(:Student :Teacher)`. So if we assign a person to a breakout room, the reasoner will infer that this person is both a student and teacher, which is not what we want. We fixed this by simply changing the intersection to a union where appropriate.

5.1.5 Timestamp

A last significant change we made was to change the datatype of some properties from `xsd:timestamp` to `xsd:string`. This is again a technical issue. This time, the issue is with the mapping. We unfortunatly couldn't find a way to map the timestamps in the SQL database to `xsd:timestamp`. We had similar problems for `xsd:dateTime` but we found a way to map these in time.

5.2 R2RML

This section outlines some of the choices we made in creating the R2RML mapping for our Virtual University ontology. The ERD in Chen's notation was the foundation for creating our database. After having issues with MariaDB in conjunction with Ontop, we decided to focus on an SQLite database that could easily store our test data. The database was created and populated using the `database.sql` file, found in the accompanying repository with this report. The R2RML mapping file, `vu-mapping.ttl`, is read by the R2RML implementation `r2rml-fat.jar`, outlined in *R2RML*, by Prof. Dr. Debruyne [3].

5.2.1 Typical R2RML mapping patterns

We first discuss the typical patterns in the mapping by way of example. Afterwards, we illustrate how we mapped to triples that were derived or otherwise computed from values in the database.

Direct mapping from database to ontology

When the database structure and the ontology have a direct relationship that can be mapped very easily, like `tbl_Courses` and `:Course`, we map them directly, as shown in listing 5.

```
CREATE TABLE IF NOT EXISTS "tbl_Courses" (
  "col_id"  integer UNIQUE,
  "col_coursenumber"  varchar(30) UNIQUE,
  "col_name"  varchar(30),
  "col_ects_credits"  integer,
  "col_grading_information"  varchar(255),
  PRIMARY KEY("col_id" AUTOINCREMENT)
);
```

Listing 5: SQL DDL for Courses table

```
<#Courses> a rr:TriplesMap;
  rr:logicalTable [rr:tableName "tbl_Courses"];

  rr:subjectMap [
    rr:template
     ↪  "http://onto.virtual-university.org
     ↪  #course_{col_coursenumber}" ;
    rr:class :Course;
  ];

  rr:predicateObjectMap [
    rr:predicate :courseCode;
    rr:objectMap [rr:column
     ↪  "col_coursenumber"];
  ].
```

Listing 6: R2RML mapping for Courses class

Mapping of subclasses

We demonstrate the mapping of subclasses using the `Teachers` example as illustrated in section 3. First, we denote the data type of all `Teachers` by distinguishing between `GuestLecturers`, `TeachingAssistants`, and `Professors`. To obtain the distinction, we construct a mapping for each of these subtypes. In listing 7 the mapping for `Professors` is shown.

```
<#ProfessorsView> rr:sqlQuery """

select p.col_username as col_username
from tbl_teachers t, tbl_persons p,
↪  tbl_teachertypes tt
where t.col_person = p.col_id and
↪  t.col_teachertype = tt.col_id
and tt.col_teachertype = 'Professor'

""".

<#Professors> a rr:TriplesMap;
  rr:logicalTable <#ProfessorsView>;

  rr:subjectMap [
    rr:template
    ↪  "http://onto.virtual-university.org
    ↪  #person_{col_username}";
    rr:class :Professor;
  ].
```

Listing 7: R2RML mapping for Professors class

First, a `ProfessorsView` is used to get all the `Teachers` of type `Professor`. Their `username`, which can be found in the database table `tbl_Persons`, is used to create the URI in the `rr:template`. All persons, namely `Teachers` and `Students`, will use the same template as URI, in order to have them identified uniquely.

Next, in listing 8, the mapping for the class `Teachers` is shown.

```
<#TeachersView> rr:sqlQuery """

select p.col_username as col_username,
↪  t.col_employee_id as col_employee_id,
↪  t.col_id as col_teacher_id
from tbl_teachers t, tbl_persons p
where t.col_person = p.col_id

""".

<#Teachers> a rr:TriplesMap;
  rr:logicalTable <#TeachersView>;

  rr:subjectMap [
    rr:template
    ↪  "http://onto.virtual-university.org
    ↪  #person_{col_username}" ;
  ];

  rr:predicateObjectMap [
    rr:predicate :employeeId;
    rr:objectMap [rr:column "col_employee_id"];
  ].
```

Listing 8: R2RML mapping for Teachers class

One can remark that the `subjectMap` has no mapping on the type of a `Teacher`. This is because a `Teacher` is always one of its subclasses, namely `GuestLecturer`, `TeachingAssistant`, or `Professor`, and the class of the individual when extracting data from the database is one of its subclasses. The employee identifier for a `Teacher` is mapped using a direct mapping from the database column `tbl_Teachers.col_employee_id` onto the property `:employeeId`.

Note that `tbl_Teachers.col_id` is also part of the select clause in `TeachersView`, but not used in the `Teacher` mapping. We will need this column later on to make a join between two classes.

Lastly, we have to map the `Persons` class. It is a direct mapping from the database. We used the friend-of-a-friend implementation of the `Persons` class, a example object map is shown:

```
rr:predicateObjectMap [
    rr:predicate foaf:nick;
    rr:objectMap [ rr:column "col_username" ];
  ];
```

Mapping of relations

By way of example we show how a `Class` is linked to its `Course` by the predicate `:partOfCourse`. This is a one-to-many relationship, since al `classes` will be linked to just one `course`. The easiest way to do this is creating an `objectMap` using a `joinCondition` between two `TripleMaps`. This is shown in listing 9

```
<#Classes>
  rr:predicateObjectMap    [
    rr:predicate :partOfCourse;
      rr:objectMap [
        rr:parentTriplesMap <#Courses>;

        rr:joinCondition [
          rr:child "col_course";
          rr:parent "col_id";
      ];
    ];
  ].
```

Listing 9: R2RML mapping for Classes joining on Course

A alternative way of mapping relationships is through a view, such is the case for `Attendences`, as shown in Listing 10

```
<#AttendancesView> rr:sqlQuery """

select p.col_username as col_username,
↪  a.col_classid as col_classid
from tbl_Attendance a, tbl_Students s,
↪  tbl_Persons p, tbl_Classes c
where a.col_studentid = s.col_id and
↪  s.col_person = p.col_id
and c.col_id = a.col_classid
```

```
""".

<#Attendances> a rr:TriplesMap;
  rr:logicalTable <#AttendancesView>;

  rr:subjectMap [
    rr:template
    ↪  "http://onto.virtual-university.org
    ↪  #person_{col_username}" ;
  ];

  rr:predicateObjectMap [
    rr:predicate :attendsClass;
    rr:objectMap [
      rr:template
      ↪  "http://onto.virtual-university.org
      ↪  #class_{col_classid}";
    ];
  ];
.
```

Listing 10: R2RML mapping for Attendances class

The `AttendencesView` is selecting the values in `tbl_Persons.col_username` and `tbl_Classes.col_classid`. When these two are linked in the `AttendencesView`, we can say that the student with the selected username attends the class with the selected `classid`. The `subjectMap` and `predicateObjectMap` are identifying these objects by their URI.

Mapping of computed values

Calculated or computed values are values present in the ontology but not present in the database. These values are calculated in the SQL query that is used as the View for the mapping. Listing 11 shows the calculation of the datatype property `numberOfStudents`, which is defined in the ontology as shown in Listing 12.

```
<#ClassAttendancesView> rr:sqlQuery """

SELECT c.col_id, COUNT(a.col_studentid) AS
↪  col_attendance
FROM tbl_Attendance a JOIN tbl_Classes c ON
↪  a.col_classid = c.col_id
GROUP BY c.col_id

""".

<#ClassAttendances> a rr:TriplesMap;
  rr:logicalTable <#ClassAttendancesView>;

  rr:subjectMap [
    rr:template
    ↪  "http://onto.virtual-university.org
    ↪  #class_{col_id}" ;
  ];

  rr:predicateObjectMap [
    rr:predicate :numberOfStudents;
    rr:objectMap [ rr:column "col_attendance" ]
    ↪  ;
  ].
```

Listing 11: R2RML mapping for class attendance

```
:numberOfStudents
  a owl:DatatypeProperty ,
  owl:FunctionalProperty ;
  rdfs:label "number of students"@en ;
  rdfs:comment "Number of students in a
  ↪  class."@en ;
  rdfs:domain :Class ;
  rdfs:range xsd:integer .
```

Listing 12: Number of students predicate in ontology

5.3 SHACL

For the assignment, we had the option to investigate a specialised topic from the class. We investigated validation of our data graph through the use of the Shapes Constraint Language SHACL[6]. The RDF framework allows anyone to say anything about anything. Two consequences following from this are that RDF cannot prevent anyone from making nonsensical or inconsistent assertions, and that RDF cannot assert that it has all information. Suppose that someone adds a triple to our triple store asserting that *'the number of seats in some physical classroom is -55.55'*, or that *'the class end date is 14/11'*, while also asserting that *'the class start date is 15/11'*. These assertions are clearly absurd. But nothing in RDF will prevent or even alert us to these absurd assertions. Even worse, a reasoner may, if the assertions don't cause any inconsistencies, infer more absurd facts from these assertions. Someone may also forget to add all information. Suppose we add assertions stating that 'x is a poll', 'x has poll question…', but forget to add an assertion that relates this poll to a Big Blue Button conference. Without this last assertion, these facts are almost useless.

From these consequences of RDF, the need for applications to validate information to detect absurd and inconsistent assertions, or incomplete information arises. To validate our data graph, we made use of SHACL. SHACL is a specification for a constraint language that allows to validate a graph through the use of so-called shapes. SHACL specifies two types of shapes: node shapes and property shapes. Node shapes contain the constraints directly on nodes, and property shapes contain the constraints on the values on a node linked through a path, e.g. a property. Recall that a node defined in RDF terms is the set of subjects and objects of triples in the graph. A SHACL engine/processor implementing the SHACL specification takes two inputs: a graph containing the RDF data to validate, which we call the data graph, and a graph containing the shapes used to validate the data, which we call the shapes graph. The SHACL engine will then output a validation report notifying of any violation, or may give some warning or information. For our project we made use of SHACL Playground[1] implemented by Holger Knublauch of TopQuadrant to validate our data graph using the shapes graph we created.

[1] https://shacl.org/playground/

This shapes graph is contained in the file `shacl.ttl`. For the rest of this section we'll summarise the more interesting shapes and constraints used in the shapes graph. All used shapes and constraints are available in the previously mentioned file.

5.3.1 Node shape and property shape

Listing 13 shows a simple shape used to validate that a student has exactly one student number, and that the value of student number is a valid string. The first triple specifies that `:StudentShape` is a node shape. The second triple specifies that this shape targets all nodes which are of the class `:Student`. We then use `sh:property` to specify that all nodes targeted by this shape should have the given property shape. We do not explicitly declare that the blank node assigned to `sh:property` is a property shape, as we can infer this from the range of `sh:property`. The first property shape validates that the targeted node must have exactly one student number. The property shape uses `sh:path` to declare a path that goes from the targeted node, to some focus node. Here, a literal value assigned to the data property `:studentNumber`. It then uses `sh:minCount 1` to check that there is at least one focus node reachable through this path, and `sh:maxCount 1` to check that there is at most one focus node reachable through this path. We use `sh:description` to provide a human readable description for this property shape that may be used in applications, and `sh:message` is used to provide a human readable message when this property shape is violated. The second node shape does a similar thing but instead checks if the focus node conforms to the `xsd:string` datatype, thus checking if the focus node is a valid string.

```
:StudentShape
  a sh:NodeShape ;
  sh:targetClass :Student ;
  sh:property [
    sh:description "Student number is required
    ↪  and limited to one value."@en ;
    sh:path :studentNumber ;
    sh:message "Student number is missing or
    ↪  too many student numbers are
    ↪  specified."@en ;
    sh:minCount 1 ;
    sh:maxCount 1 ] ;
  sh:property [
    sh:description "Student number must be a
    ↪  valid string."@en ;
    sh:path :studentNumber ;
    sh:message "Student number is not a valid
    ↪  string."@en ;
    sh:datatype xsd:string ] .
```

Listing 13: Simple student validation shape

5.3.2 Severity

Any violations of the node and property shapes are by default of the *Violation* severity. Sometimes we may wish to instead warn or inform the user of some violation. For this purpose we can make use of the predicate `sh:severity`, which may be assigned the values `sh:Violation`, `sh:Warning` or `sh:Info`. These will alter the validation report and can be used by the application. Listing 14 gives an example of a shape targeting the `Person` class, where the shown property shape validates that a first name and family name are present. If not present, this would instead be considered a warning and not a violation. Note that we have excluded the rest of the node shape for brevity.

```
:PersonShape
  a sh:NodeShape ;
  sh:targetClass foaf:Person ;
  sh:property [
    sh:description "First name and family name
    ↪  are highly advised."@en ;
    sh:path foaf:firstName, foaf:familyName ;
    sh:severity sh:Warning ;
    sh:message "First name or family name are
    ↪  missing but advised."@en ;
    sh:minCount 1 ] ;
  ...
```

Listing 14: First and family name validation

5.3.3 Pattern Constraint

In some cases we may wish to validate a string literal value against a pattern. This can be achieved through the use of `sh:pattern`, which takes a regex pattern. Listing 15 gives an example of a node shape in our shape graph targeting the `:Person` class, where the shown property shape validates that the email is well-formed using a regex.

```
:PersonShape
  a sh:NodeShape ;
  sh:targetClass foaf:Person ;
  ...
  sh:property [
    sh:description "Email must be
    ↪  well-formed."@en ;
    sh:path foaf:mbox ;
    sh:message "Email is not well-formed"@en;
    sh:pattern "^([a-zA-Z0-9_\\-\\.]+)
    ↪  @([a-zA-Z0-9_\\-\\.]+)
    ↪  \\.([a-zA-Z]{2,5})$" ;
    sh:datatype xsd:string ] ;
  ...
```

Listing 15: Email validation through RegEx

5.3.4 Value Range Constraint

If the focus node is a numerical literal, we may wish to check if its value is in some range. SHACL provides several constraints for this. Listing 16 gives an example of a node shape in our shape graph targeting the `:PhysicalClassroom` class, where the shown property shape validates that the number of seats is greater than 0. This is done using `sh:minExclusive`.

```
:PhysicalClassroomShape
  a sh:NodeShape ;
  sh:targetClass :PhysicalClassroom ;
      ...
  sh:property [
```

```
    sh:description "Number of seats must be a
    ↪  valid integer." ;
    sh:path :numberOfSeats ;
    sh:message "Number of seats is a invalid
    ↪  integer."@en ;
    sh:datatype xsd:int ;
    sh:minExclusive 0 ] ;
  ...
```

Listing 16: Range validation of number of seats

5.3.5 Property Pair Constraint

SHACL also provides constraint to compare properties with each other pair-wise. This could for example be used to check that some start date is chronologically before an end date. Listing 17 gives an example of a node shape in our shape graph targeting the `:Class` class, where the shown property shape validates that the start date of a class is before the end date of that same class. This is done using `sh:lessThanOrEquals`.

```
:ClassShape
  a sh:NodeShape ;
  sh:targetClass :Class ;
    ...
  sh:property [
    sh:description "Class start date must be
    ↪  before class end date."@en ;
    sh:path :classStartDate ;
    sh:message "Class start date is not before
    ↪  class end date."@en ;
    sh:lessThanOrEquals :classEndDate ] ;
  ...
```

Listing 17: Property pair constraint on start and end date

5.3.6 Advanced Property Paths

In some instances the default `sh:path` may not be powerful enough. Suppose we wish to express that the focus node of a property shape should be the node we reach using one property `example:property_1`, followed by another property `example:property_2`, or suppose we want to reach all nodes through the transitive closure of a property. For this purpose SHACL provides ways to define more complex paths. Listing 18 gives an example of a node shape in our shape graph targeting the `:Recording` class, where we the shown property shape validates that there is either a class, a classroom, or both assigned to this recording. This is done using `sh:alternativePath` and `sh:minCount`.

```
:RecordingShape
  a sh:NodeShape ;
  sh:targetClass :Recording ;
  ...
  sh:property [
    sh:description "A recording should be
    ↪  assigned to at least one Class or
    ↪  Classroom" ;
    sh:path [sh:alternativePath
    ↪  (:recordingForClass
    ↪  :recordingForClassroom)] ;
    sh:message "Recording is not assigned to at
    ↪  least one Class or Classroom" ;
    sh:minCount 1 ] .
```

Listing 18: Advanced property paths on Recording class

5.3.7 Validation Report

Lastly we'll give an example of a validation report using hypothetical data. Suppose we wish to register a new student, we add the triples specified in listing 19. It is clear that this information is incomplete and not correct. The email address is not well-formed and the student number and nickname are missing. We've also forgotten to add a first name to this student. This is not a violation, but this still leaves the information incomplete. Using SHACL Playground we run our shapes graph against this data graph, resulting in the validation report in listing 20. From this validation report we find 4 results. The first is a warning that we might have forgotten to add a first name or family name. Next are three violations that the email is not well-formed, a nickname is missing and a student number is missing. In this validation report we also see for each result which focus node caused the warning or violation. For this project we made use of SHACL Playground as it is easy, but it is clear that for bigger projects and production environments this is not feasible. APIs do exist for SHACL in several languages, such as .NET or Scala.

```
:Jim
  a :Student ;
  foaf:familyName "Foobarson" ;
  foaf:mbox "Jim" .
```

Listing 19: Example incomplete student

```
[
  a sh:ValidationResult ;
  sh:resultSeverity sh:Warning ;
  sh:sourceConstraintComponent
  ↪  sh:MinCountConstraintComponent ;
  sh:sourceShape _:n5323 ;
  sh:focusNode
  ↪  <http://onto.virtual-university.org#Jim>;
  sh:resultPath foaf:firstName ;
  sh:resultMessage "First name or family name
  ↪  are missing but advised."@en ;
] .
[
  a sh:ValidationResult ;
  sh:resultSeverity sh:Violation ;
  sh:sourceConstraintComponent
  ↪  sh:PatternConstraintComponent ;
  sh:sourceShape _:n5325 ;
  sh:focusNode
  ↪  <http://onto.virtual-university.org#Jim>;
  sh:value "Jim" ;
  sh:resultPath foaf:mbox ;
  sh:resultMessage "Email is not
  ↪  well-formed."@en ;
```

```
] .
[
  a sh:ValidationResult ;
  sh:resultSeverity sh:Violation ;
  sh:sourceConstraintComponent
  ↪  sh:MinCountConstraintComponent ;
  sh:sourceShape _:n5326 ;
  sh:focusNode
  ↪  <http://onto.virtual-university.org#Jim>;
  sh:resultPath foaf:nick ;
  sh:resultMessage "Nickname is missing or too
  ↪  many nicknames are specified."@en ;
] .
[
  a sh:ValidationResult ;
  sh:resultSeverity sh:Violation ;
  sh:sourceConstraintComponent
  ↪  sh:MinCountConstraintComponent ;
  sh:sourceShape _:n5328 ;
  sh:focusNode
  ↪  <http://onto.virtual-university.org#Jim>;
  sh:resultPath
  ↪  <http://onto.virtual-university.org
  ↪  #studentNumber> ;
  sh:resultMessage "Student number is missing
  ↪  or too many student numbers are
  ↪  specified."@en ;
] .
```

Listing 20: SHACL validation report

5.4 Linked Data Frontend

The assignment tasked us with setting up a linked data frontend for our Virtual University ontology. We took inspiration from the lab sessions and opted to use an Apache Jetty web server and servlet container to host an instance of Apache Jena Fuseki2. Ontology documentation was provided through WIDOCO[5].

The entire linked data frontend is realized through Docker, specifically using a network of containers orchestrated through `docker-compose`. Section 7 goes into greater detail about the set-up of our front-end.

5.4.1 Jetty and Jena Fuseki

As mentioned previously, the Virtual University SPARQL endpoint is hosted by Jena Fuseki2, inside Apache Jetty. The endpoint provides access to over 3000 triples, which are realized through the R2RML mapping outlined in section 5.2.

Originally, we chose to set up an ETL pipeline with Ontop and MariaDB, but we ran into several issues which cost us many hours and led us to revert back to Fuseki2 to host our SPARQL endpoint.

Apache Jetty and Jena Fuseki2 were set up using a handcrafted Dockerfile, which includes a working reasoner and overrides to the Fuseki configuration file, to allow for execution in a server environment. Because of a lack of complete ETL pipeline, the user is required to manually upload RDF triples to the endpoint, prior to querying it.

5.4.2 WIDOCO

The Virtual University ontology documentation is realized using WIDOCO[5]. A handcrafted Docker container with the necessary execution steps was created to facilitate complete autonomous documentation generation.

6 Demonstrator

Part of the assignment included the suggestion of realizing a non-trivial demonstrator to motivate the usage of our Virtual University ontology. We present a Virtual University web application as a proof-of-concept in this section.

HTTP requests made by the web application to the SPARQL endpoint adhere to the format described by Feigenbaum et al. in section 2.1.3: *query via POST with URL-encoded parameters* in the *SPARQL 1.1 Protocol*[4]:

```
POST /fuseki/virtual-university/sparql HTTP/1.1
Host: localhost:8080
Accept: application/sparql-results+json
Content-Type: application/x-www-form-urlencoded
```

The information is queried, retrieved, and subsequently parsed according to the JSON format specification *SPARQL 1.1 Query Results JSON Format*[8] by the web application to hydrate the data in the application.

6.1 Virtual University Web Application

The web application acts as a quick reference for teachers and administrators in the university or learning organisation that employ our ontology for tracking online learning. Specifically, it features the following functionality through querying of the SPARQL endpoint:

- List all teachers in the underlying database.
- Load courses for a given teacher.
- Display summarized information for a given course, such as which platforms were used to host classes.

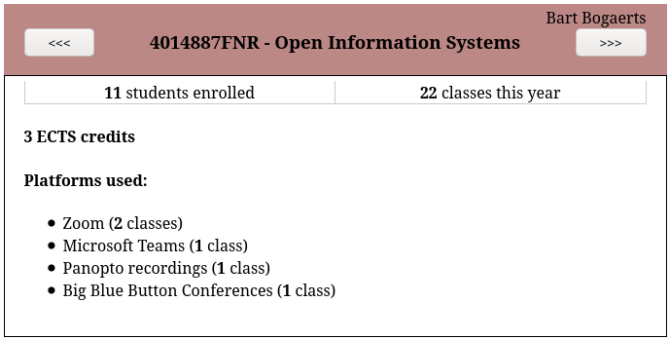

Figure 14: Summary of the selected course

6.1.1 Student engagement through charts

Additionally, teachers can visualize student engagement numbers recorded in the ontology through four interactive charts.

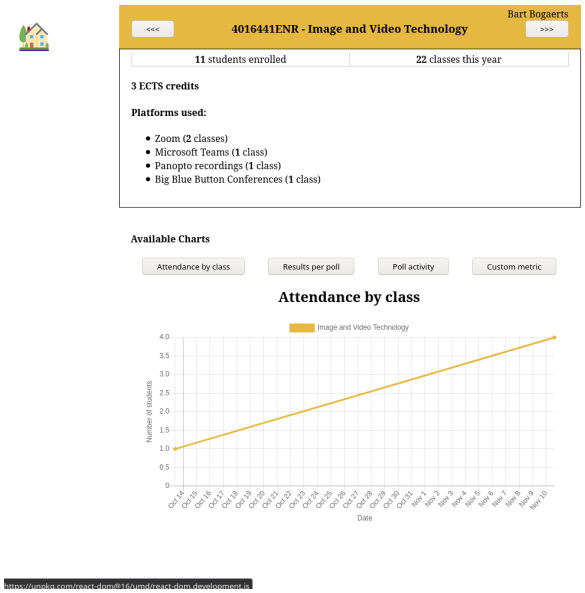

Figure 15: Main view when a teacher is selected

6.1.2 Best buddies

As a demonstration of the inherent expressiveness of SPARQL, we opted to feature a *best buddies* section on the landing page of the application, where the student and teacher with the most interactions are proudly displayed. This functionality is realized by the following SPARQL query, and serves as leading example for the SPARQL queries used throughout the rest of the application:

```
SELECT ?teacherFirstName ?teacherLastName
↪  ?studentFirstName ?studentLastName
↪  ?encounters
WHERE {
  {
    SELECT ?teacherFirstName ?teacherLastName
    ↪  ?studentFirstName ?studentLastName
    ↪  (COUNT(*) AS ?encounters)
    WHERE {
      ?class :classTaughtBy ?teacher.
      ?student :attendsClass ?class.
      ?student foaf:firstName
      ↪  ?studentFirstName.
      ?teacher foaf:firstName
      ↪  ?teacherFirstName.
      ?student foaf:familyName ?studentLastName
      ↪  .
      ?teacher foaf:familyName ?teacherLastName
      ↪  .
    }
    GROUP BY ?teacherFirstName ?teacherLastName
    ↪  ?studentFirstName ?studentLastName
  } FILTER(?encounters = ?maxencounters) {
    SELECT (MAX(?encounters2) AS
    ↪  ?maxencounters)
    WHERE {
      {
        SELECT ?teacher ?student (COUNT(*) AS
        ↪  ?encounters2)
        WHERE {
          ?class :classTaughtBy ?teacher.
          ?student :attendsClass ?class.
        }
        GROUP BY ?teacher ?student
      }
    }
  }
}
```

Listing 21: Best buddy SPARQL query

7 Artifacts and Resources

7.1 Files and code

This reports comes with a number of artifacts including source code, a test database, generated RDF tuples, etc. The artifacts, along with a copy of this report, can be found by navigating to the following git repository, hosted on GitHub.

https://github.com/decrn/virtual-university

You may choose to download a zipped copy of the repository by pressing the green *Code* button at the top right, and selecting *Download ZIP*.

7.2 Set up

As mentioned in section 5.4 on the linked data frontend, our services are started through a network of Docker containers, orchestrated through a `docker-compose.yaml` file. Naturally, this dictates the requirement for Docker[2] and `docker-compose`[3] to be installed on your system.

To run this project, navigate to your local copy of the project files provided with this report, ensuring you are in the directory with the `docker-compose.yaml` file. Next, start the services by executing the following command:

```
docker-compose up --build
```

Following this, services should become available to your host computer. Navigate to http://localhost:8080/fuseki and upload the provided `virtual-university.ttl` and `vu-data.ttl` files to the `virtual-university` dataset already present in the web interface.

Verify the fuseki dataset has been populated by pointing your browser to http://localhost:8081. You should be greeted with the Virtual University web application containing a number of clickable teachers as seen in figure 16.

Figure 16: Homepage of the Virtual University web application with clickable teachers

[2]Install Docker https://docs.docker.com/get-docker/
[3]Install docker-compose https://docs.docker.com/compose/install/

WIDOCO

Ontology documentation generated by WIDOCO can be found in the `data/public/` directory, following a succesful `docker-compose up --build` command. Should you prefer to view the documentation through a web server, rather than offline, you may visit http://localhost:8082/index-en.html, provided the container is running.

Troubleshooting

At any point, services may be restarted using the following command:

```
docker compose down -v && \
docker-compose up --build
```

If any of the containers fail to start, ensure the following TCP/IP ports are unbound:

- `:8080`, for Apache Jetty and Apache Jena Fuseki2
- `:8081`, for the Virtual University web application
- `:8082`, for the WIDOCO ontology documentation

8 Conclusion

Because of the COVID-19 pandemic, many universities and other educational institutes hold lectures and meetings online. Because of this, an enormous wave of information is created every day that needs to be handled and managed in the correct way. In our report, we proposed the creation of an open information system that may help to tackle this issue. Furthermore, we outlined the development and design of such a system, taking special care to motivate the design choices we made. Based on our conceptual model, and at the heart of this system is the ontology. In subsequent sections we provided ways to implement and use our ontology through the use of a demonstrator and simple proof-of-concept. As the ontology cannot guarantee that incomplete, inconsistent or absurd information is added, we also provided ways to validate any data that is to be added, through the use of SHACL, a constraint language. At the end of the report, we documented how to reproduce our system, and how to access the created artifacts.

9 References

[1] DBpedia Association. *About:work*. URL: `http://dbpedia.org/ontology/Work`.

[2] Dan Brickley and Libby Miller. *FOAF Vocabulary Specification 0.99*. URL: `http://xmlns.com/foaf/spec/`.

[3] Christophe Debruyne. *R2RML*. `https://github.com/chrdebru/r2rml`. 2013.

[4] Lee Feigenbaum et al. *SPARQL 1.1 Protocol*. W3C Recommendation. W3C, Mar. 2013. URL: `https://www.w3.org/TR/sparql11-protocol/`.

[5] Daniel Garijo. "WIDOCO: a wizard for documenting ontologies". In: *International Semantic Web Conference*. Springer, Cham. 2017, pp. 94–102. DOI: `10.1007/978-3-319-68204-4_9`. URL: `http://dgarijo.com/papers/widoco-iswc2017.pdf`.

[6] Dimitris Kontokostas and Holger Knublauch. *Shapes Constraint Language (SHACL)*. W3C Recommendation. W3C, July 2017. URL: `https://www.w3.org/TR/2017/REC-shacl-20170720/`.

[7] *Owl web ontology language guide*. URL: `https://www.w3.org/TR/owl-guide/`.

[8] Andy Seaborne. *SPARQL 1.1 Query Results JSON Format*. W3C Recommendation. W3C, Mar. 2013. URL: `https://www.w3.org/TR/sparql11-results-json/`.

A Conceptual schema