Academic Data Profiling and Cleaning: A University Timetable Case Study

Alexandre Lemos, Helena Galhardas, Inês Lynce and Pedro T. Monteiro
INESC-ID and Instituto Superior Técnico, Universidade de Lisboa

Abstract

FénixEdu™ is the information system used at Instituto Superior Técnico (IST) to store and manage data related to academic life. Currently, IST does not provide an automatic procedure to generate timetables. The data, required to fully automate the generation of timetables, is available through an API of the Fénix system. However, this data has data quality problems that range from approximate duplicates to integrity constraint violation. Therefore, before generating timetables it is important to clean the data.

This work has two goals: i) to assess the quality of the curriculum data of IST, and ii) to improve the quality of the data by cleaning it. We used two software tools to profile data: Pentaho Data Integration and Open Source Data Quality and Profiling. The data quality problems range from approximate duplicates to violations of functional dependencies. The data cleaning prototype named Cleenex was used to clean all data quality problems found during the profiling phase. The data cleaning procedure removed the 7% and 2% of exact and approximated duplicates respectively. All heterogeneous date representations were converted to a standard format. Furthermore, all redundant or incomplete data (14%) were removed.

1 Introduction

Nowadays, at Instituto Superior Técnico (IST), the generation of timetables is at a turning point as the university has decided recently to move from handmade solutions to an automatic system [4, 9, 10, 12, 14]. The timetables for the 2019/2020 semester were still generated by hand, based on the timetable used in the previous year and thus minimizing the number of modifications performed. It is expected that the timetables will be automatically generated in the near future.

In order to automatically generate timetables, it is necessary to obtain the data corresponding to the degrees, courses, rooms as well as their characteristics. Moreover, the timetables from the previous years are also important as one may want to start the new timetable using a previous acceptable version. FénixEdu™ is the information system used at IST to store and manage data related to academic life, including all the necessary data to automatically generate timetables. This data can be extracted, in JSON format, through the system's public API[1].

However, it is known that the data stored in FénixEdu™ has some data quality problems. For example, the course names *"Introdução à Investigação em Engenharia Electrotécnica e de Computadores"* and *"Introdução à Investigação em Engenharia Electrotécnica e de Computadore"*, correspond to the same course although there is an *"s"* missing in the second name.

In order to correctly generate timetables, it is important to first clean data quality problems that exist in the data stored at FénixEdu™. Considering the example above, the resulting timetable is different if one considers the two forms of course names as two different courses or as a single course. To be able to effectively clean all the data quality problems existing in the data extracted, it is important to first systematically profile the data in order to collect all the existing data quality problems.

Moreover, not all relevant information is directly obtained from the system. In fact, some data useful to generate timetables is obtained as a result of applying data transformations to the data extracted from Fénix. For example, the information about the periodicity of the lectures to be scheduled is important when one is generating the weekly timetables. However, the FénixEdu database contains only the lectures as separate events. The periodicity can only be obtained by merging all lectures that occur in the same weekday, time frame and room.

[1] http://fenixedu.org/dev/api/

1.1 Objectives

The goal of this work is to collect all the existing data quality problems, from the FénixEdu database, and then correctly clean the data to be used in the generation of automatic timetables.

1.2 Data Profiling, Transformation and Cleaning Process

In this section, we present the data profiling, transformation and cleaning process that needs to be applied to the data extracted from the FénixEdu system. Furthermore, we discuss the advantages and shortcomings of the tools used to perform the two tasks: data profiling and data cleaning.

The steps performed in the context of this work are organized according to the workflow represented in Figure 1. The first step, (a) Converting JSON to relational, is the step where the data extracted from the FénixEdu^{TM} system is converted from JSON to relational format. This step is particularly important since one of the data profiling tools chosen to be used (see below) does not support the JSON format as input. Also, the data cleaning prototype chosen to be used only supports relational data. Step (b) Data Profiling focuses on assessing the quality of the data output by the data conversion in step (a). This step produces a report containing all the data quality problems found. Step (c) Data Cleaning is the step where the data converted in step (a) is cleaned based on the findings in step (b). The last step (d) Data Transformation transforms the output of step (c) in order to obtain data in the format required for automating the production of timetables named *ectt*. The standard format *ectt*[2] exists for curriculum-based course timetabling [3]. However, this format has limitations and it is not adequate to be used by all the data obtained FénixEdu^{TM}. This format considers that all lectures have the same duration and does not enable the storage of the timetabling history (the timetables from past semesters). Therefore, we proposed a data model that the cleaned data should satisfy (present in Appendix A). The data resulting from the data profiling, transformation and cleaning process is stored in a relational database whose schema is presented in Appendix A.

To address the first step (a) converting JSON to relational format, we first used Open-Refine [8]. However, the program built could not finish loading in a reasonable amount of time when opening multiple JSON files. So, we decided to use the Pentaho Data Integration (PDI)[3] tool to perform the data conversion. Converting data from JSON to a relational schema was purely a data transformation without adding any type of constraints or pre-defined structure. The idea was to represent, as faithfully as possible, in a relational format, the JSON data obtained from FénixEdu^{TM}. Using constraints in the initial relational schema would impose a certain level of quality to the data imported from FénixEdu^{TM}. For this reason, the constraints were only considered for the relational model used to store the final cleaned data.

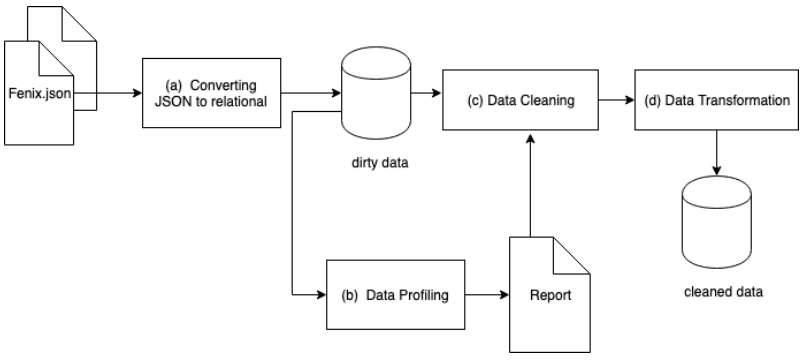


Figure 1: Data profiling, transformation and cleaning process.

The data profiling step (b) was handled by the Arrahtec's Open Source Data Quality and Profiling (OSQD)[4] and PDI (without plug-ins). OSQD provides a wide range of functionalities and it is easy to use. However, OSQD does not provide all the functionality needed. For example, the task of verifying if the room capacity is enough for the scheduled class is impossible to perform with OSQD. It only enables us to check for inclusion dependencies between fields. To fill this

[2]The *ectt* Input Format is available at http://tabu.diegm.uniud.it/ctt/index.php?page=format.
[3]PDI: Pentaho Data Integration documentation is available at http://community.pentaho.com/.
[4]Arrahtec's Open Source Data Quality and Profiling (OSQD) is available at http://www.arrahtec.com/.

gap, we use PDI. PDI provides more expressiveness than OSQD since its base functionalities can be combined in order to obtain more complex functionalities, for example, checking if the values resulting from the concatenation of the columns A and B are equal to the values of column C.

The last two steps (c) Data Cleaning and (d) Data Transformation were performed using the data cleaning prototype named Cleenex [6]. The Cleenex prototype is still under development and so this work was also used to assess the advantages and shortcomings of the tool, testing it with a real case of dirty data. The prototype successfully solved all the data quality problems found.

1.3 Organization of the Document

The structure of this document is a direct mapping of the diagram shown in Figure 1. Section 2 describes the first step: Converting JSON to relational format. In, Section 3 the data profiling method is explained and the data profiling results are discussed. Finally, Section 4 describes the process of data cleaning and transformation. Section 5 presents the conclusions of this case study. All the data cleaning programs used are available in Appendix B.

2 Converting JSON data to relational data

This section describes the conversion of JSON data into relational data. The goal of this conversion is to obtain the data in a format that can be given as input to a data profiling tool. The conversion must produce data in a format as similar as possible to the original one. This way, one avoids adding or removing data quality problems during the conversion. Therefore, all the data quality problems from the original data can be detected.

Initially, the conversion step was performed using the data wrangling tool Open-Refine [8]. Open-Refine has the capability of converting multiple JSON files into the relational format in a single and easy step. The tool analyses the JSON files in order to choose the correct relational structure to represent the input data. However, this approach did not work since the tool could not open all the necessary files. The tool has a limit for the number of files/sizes it can handle. There are 4306 (2,4 MB) JSON files corresponding to the spaces available at IST and 86 files (1,92 MB) corresponding to degree data. These files could not be processed. Therefore, these data conversions were implemented using the Extract Transform Load (ETL) tool Pentaho Data Integration (PDI).

The relational model of the database that results from the conversion step is represented in Figure 2. The *space* table contains data about all existing spaces at IST, from buildings to classrooms. This table has a foreign key (FK) to itself in order to establish a relationship between spaces (*e.g.* FA1 is a classroom of the building *Informática I*). All tables that are somehow related to *spaces* have a foreign key to this table (for example *degree*, *examRoom*, *event* and *lecture*). Note that the degrees are lectured in a specific campus, so table *degree* has a foreign key to the table *space*.

The *course* table contains data about existing courses (*e.g.*, the number of students enrolled) and is connected to table *degree* through the table *courseDegree*, thus indicating that a course can belong to multiple degrees and vice-versa. The *course* is also connected to the tables *teacher* (the teacher who teaches a course), *exam* (the evaluation of the course), *courseWorkload* (the workload that the different type of lectures must have), *bibliographicReferences* (the bibliographic recommendation of the course), *lessonPeriod* (the different periods during which is the course is lectured) and *lectures* (all lectures scheduled and their characteristics).

Table *event* contains data about all events (including lectures) for a given week. This table stores redundant data since part of the data is also present in table *lecture* which is more complete. The reason for the existence of this table is to store all the events (not only lectures) that are planned for each room. Finally, table *examRoom* represents the rooms allocated to different exams.

As mentioned above, some of these 12 tables store redundant data; recall that they were created to be as faithfully as possible to the extracted files. Additionally, no primary key, foreign key nor integrity constraints were initially imposed since this would require to clean the data before profiling it.

Figure 3 illustrates the process required to convert the data concerning courses from JSON to relational. Before detailing the reason behind the steps that compose the process, it is important to explain how the data is obtained. The FénixEdu ^{TM} system API returns a JSON file for each course. The file for each course is composed by different JSON objects as exemplified in Listing 1 for

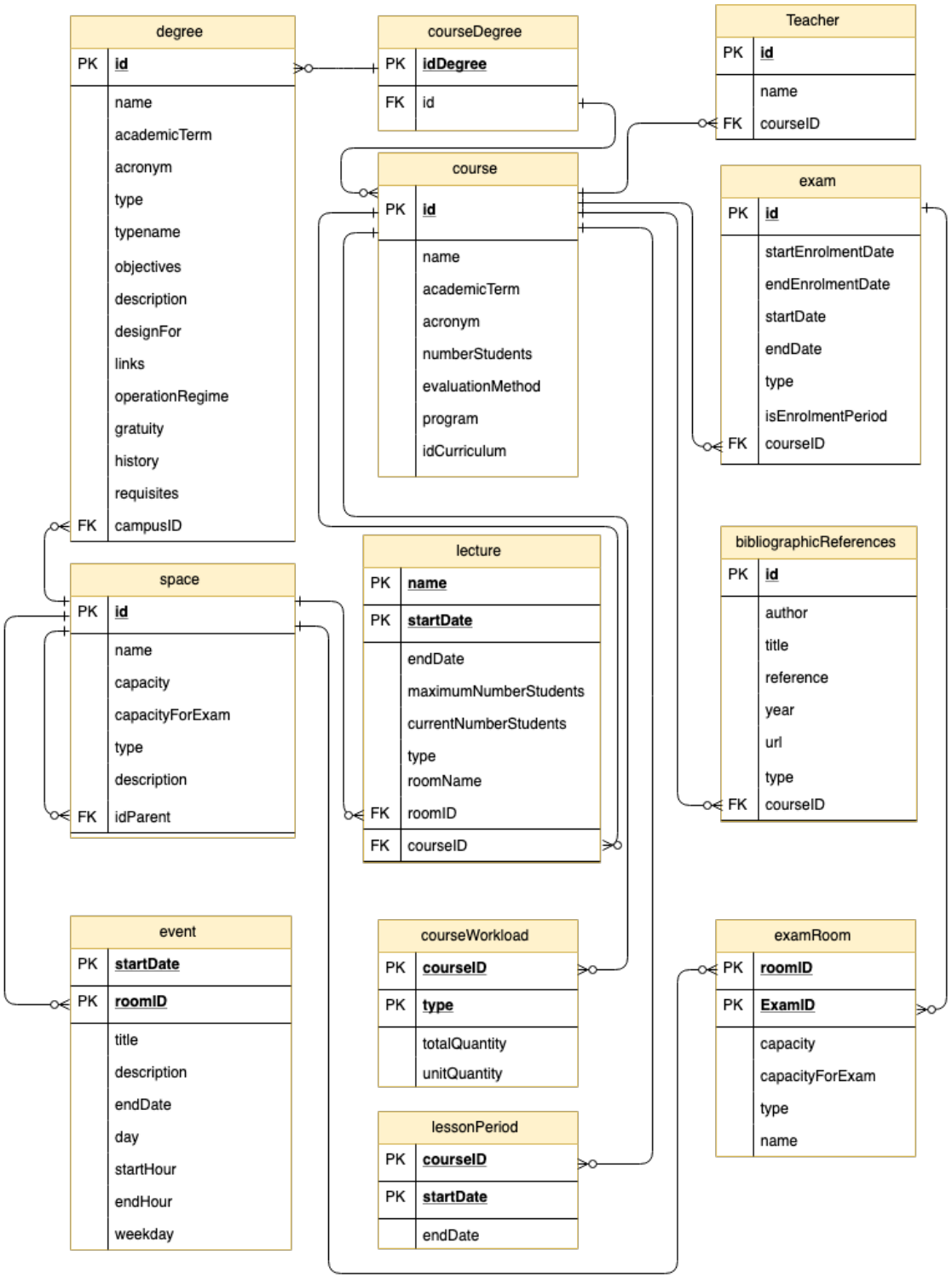


Figure 2: Relational model of the database created to mirror the data obtained from FénixEdu™. FK and PK represent the foreign and primary keys of the table, respectively. $A \rightarrowtail\!\!\!\!\!\!\bullet\ B$ represents a 1:m relationship, between table A and table B.

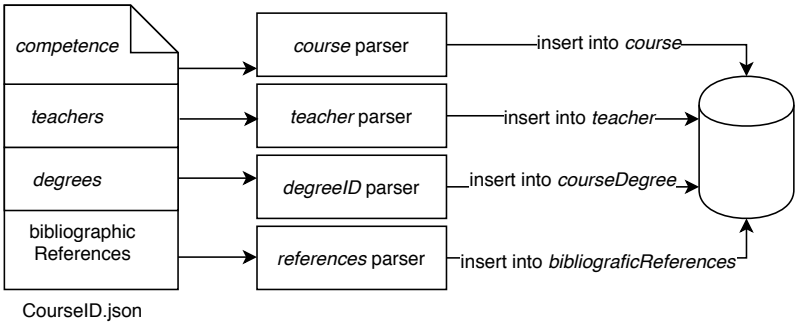


Figure 3: The data conversion process for the tables *course*, *teacher*, *courseDegree*, and *bibliographic references*.

a course. The *course* contains: (i) the fields present in lines 2 to 7, (ii) the array *competences* (lines 8-39), and (iii) the array *teachers* (lines 40-49). The array *competences* is a single JSON object which contains the *bibliographic references* array and the *degree* array (lines 26-37). This single object also contains the fields *program* (line 10) and *curriculum* id (line 9). The *degree* array stores data about the degrees (*id* and *name*) which offer this course. In the example shown in Listing 1, the course is lectured to three different degrees: *Mestrado Bolonha em Matemática e Aplicações*, *Mestrado Bolonha em Engenharia Informática e Computadores - Alameda* and *Mestrado Bolonha em Engenharia Informática e Computadores - Taguspark*. The *teachers* array contains all data related to the course teachers.

Listing 1: Example of JSON file for a course.

```
1  {
2    "acronym" : "ALC95",
3    "name" : "Algoritmos para Lógica Computacional",
4    "academicTerm" : "1 Semestre 2017/2018",
5    "evaluationMethod" : "50% Exame + 50% Projetos",
6    "numberOfAttendingStudents" : 30,
7    "url":"...",
8    "competences" : [ {
9      "id" : "1972553925001218",
10     "program" : "...",
11     "bibliographicReferences" : [ {
12       "author" : "A. Biere, M. Heule, H. van Maaren and T. Walsh",
13       "reference" : "IOS Press",
14       "title" : "Handbook of Satisfiability",
15       "year" : "2009",
16       "type" : "MAIN",
17       "url" : null
18     }, {
19       "author" : "F. Van Harmelen, Vladimir Lifschitz, and Bruce Porter",
20       "reference" : "Elsevier",
21       "title" : "Handbook of Knowledge Representation",
22       "year" : "2008",
23       "type" : "SECONDARY",
24       "url" : null
25     } ],
26     "degrees" : [ {
27       "id" : "2761663971473",
28       "name" : "Mestrado Bolonha em Mateática e Aplicações",
29       "acronym" : "ALC"
30     }, {
31       "id" : "2761663971475",
32       "name" : "Mestrado Bolonha em Engenharia Informática e de Computadores -
             Alameda",
33       "acronym" : "ALC"
34     }, {
35       "id" : "2761663971585",
36       "name" : "Mestrado Bolonha em Engenharia Informática e de Computadores -
             Taguspark",
```

```
37        "acronym" : "ALC"
38      } ]
39    } ],
40    "teachers" : [ {
41      "name" : "John Doe",
42      "istId" : "ist11111",
43      "mails" : [ "john.doe@tecnico.ulisboa.pt" ],
44      "urls" : [ "http://johndoe.pt/" ]
45    }, {
46      "name" : "Jane Doe",
47      "istId" : "ist19999",
48      "mails" : [ "jane.doe@tecnico.ulisboa.pt"],
49      "urls" : [ "https://fenix.tecnico.ulisboa.pt/homepage/ist19999"]
50    } ]}
```

The step *course parser*, shown in Figure 3, parses the entire JSON file obtaining all the different fields and objects described in Listing 1. Note that the *courseID* is a unique identifier for the course and is not present in any field of the JSON file. This value is obtained from the file name. After these steps are executed, it is possible to create a new record in table *course* with the information from lines 1 to 10 of Listing 1.

However, the JSON file contains additional data. The step *degreeID parser* iterates over the degree array in order to obtain all the degree where this course is lectured. After this step, it is possible to create a new record in table *courseDegree*.

The step *teacher parser* iterates over the teacher array in order to be able to perform the tuple insertions into the table *teacher*. Finally, the step *references parser* iterate over the *bibliographicReferences* array and inserts this data into the table *bibliographicReferences*.

The conversion process shown in Figure 3 was executed for all course files, transforming all data contained in the files to tables. Similar processes were executed to convert the remaining JSON files to the relational model shown in Figure 2.

3 Data Profiling

A taxonomy of data profiling tasks is presented in [1]. The data profiling tasks can be separated into three main types: single-column, multi-column and inclusion dependencies.

Single-column data profiling comprises a set of tasks for analyzing individual columns of the tables. These tasks are performed for every column in order to obtain the corresponding metadata, namely the number of incomplete entries, the number of unique values, data patterns and value distributions.

Multi-column data profiling is a set of tasks for analyzing multiple related columns at the same time. The tasks performed are correlations, clusters and outlier detection. These tasks find patterns between columns (*e.g.*, finding the tuples where the room's capacity for the lecture assigned is smaller than the number of students enrolled).

Functional dependencies correspond to constraints involving multiple columns (some required combining columns, *e.g.* checking if column A is a subset of column C). Functional dependencies are also checked against columns of different tables (*e.g.*, the data from table A is a subset of the data stored in table B).

The goal of data profiling, in this context, is to find all the data quality problems that exist in the data set extracted from Fénix, according to the data profiling tasks referred above. The section is organized in the following way. In Section 3.1, the results from the data profiling tasks are summarized. Sections 3.2 and 3.3 summarize the different data quality problems found and suggest possible data cleaning procedures, respectively. Section 3.4 describes the tools used to perform the data profiling tasks and discusses their advantages/disadvantages.

3.1 Data Profiling Results

This section summarizes the data profiling results, organized by table of the relational model represented in Figure 2.

We would like to remark that some data quality problems may arise due to the extraction process. For example, when considering courses, the corresponding JSON files were exported from the Fénix system for each available degree. This can cause the existence of duplicates since some courses are available for different degrees.

3.1.1 Degree

This section lists the data profiling results for the data contained in the table whose scheme is:

degree(id, name, academicTerm, acronym, type, typename, objectives, description, designFor, links, operationRegime, gratuity, history, requisites, campusID).
campusID: FK(space).

3.1.1.1 Single-Column Analysis

The results are summarized in the Table 1. They show that the fields required to automatically generate timetables are all non-null. The patterns and corresponding frequencies presented in the fields *type* and *typename* are shown in Figure 4.

Table 1: Single-Column analysis for the table *degree*.

field	# of non-null values	# of unique values	# of null values	pattern (frequency)
id	86	86	0	
name	86	66	0	
academicTerm	86	2	0	2015/2016 (5) 2016/2017 (81)
acronym	86	86	0	
type	86	6	0	see Figure 4.
typename	86	6	0	see Figure 4.
objectives	82	82	4	
description	73	73	13	
designFor	63	63	23	
links	41	40	45	
gratuity	41	40	45	
operationRegime	50	49	36	
history	61	60	25	
requisites	41	41	45	
campusId	86	2	0	*2448131360897* (78) *2448131360898* (8)

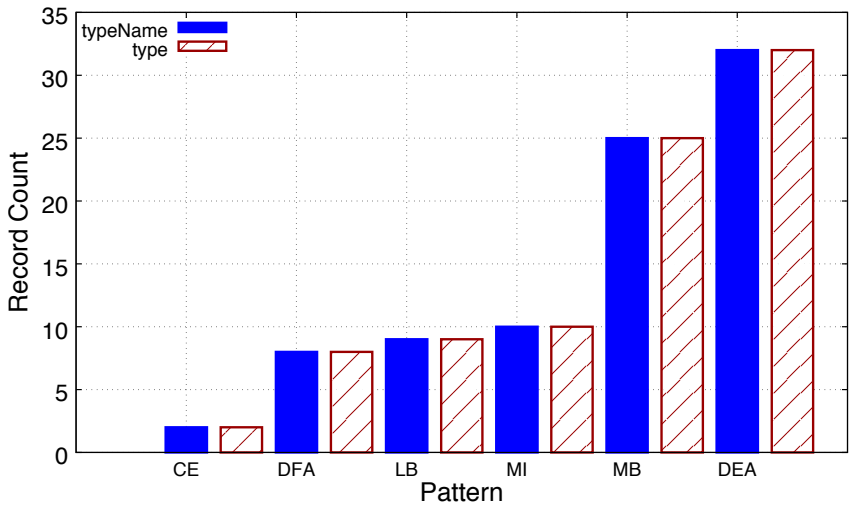


Figure 4: The frequency of patterns for the fields *type* and *typeName*.

3.1.1.2 Multi-Column Analysis

The fields *type* and *typeName* contain always the same data.

3.1.2 Course

This section lists the data profiling results for the data contained in the table whose scheme is:

course(id, name, academicTerm, acronym, numberStudents, evaluationMethod, program, curriculumID).

3.1.2.1 Single-Column Analysis

The results are summarized in the Table 2. The important facts about these results are discussed below.

Table 2: Single-Column analysis for the table *course*.

field	# of non-null values	# of unique values	# of null values	pattern (frequency)
id	5275	4811	0	
name	5275	1832	0	
academicterm	5275	4	0	1 Sem 2016-2017 (1264) 2 Sem 2016-2017 (1249) 1 Sem 2017-2018 (1418) 2 Sem 2017-2018 (1344)
acronym	5275	3799	0	
numberStudents	5275	323	0	
evaluationMethod	1386	1111	3889	
program	2336	1490	2939	
curriculumID	4401	1654	874	

id: According to the university specification the course *id* should be the unique identifier of the execution of a course and thus the primary key of this table. However, this field contains duplicate values. The reason for the 464 repetitions lies on the duplicates that arise from the data extraction process. The main reason for these occurrences is the fact that the same course may be lectured to different degrees.

name: Besides the existence of exact duplicates that exist (reasons explained above), there is also approximate duplicates in the *name* field. To detect approximate duplicate values we compared every name with all the other names, computing the Levenshtein distance [11] for each pair.

This process identifies many false positives, like: *Electrónica I; Electrónica II*. It also identifies: *Introdução à Investigação em Engenharia Electrotécnica e de Computadore; Introdução à Investigação em Engenharia Electrotécnica e de Computadores* (which is a true positive). However, both have the same Levenshtein distance 1.

evaluationMethod, program, and curriculumID: The null values can be explained by the existence of courses such as *Opção* (option in English) which are fake courses. These courses are only place holders to symbolize the possibility of choosing any course. Therefore, the fields corresponding to the detailed description of the course (the course curricular *program* and *evaluationMethod*) are null.

3.1.3 Course Workload

The table *courseWorkload*, whose schema is shown below, stores the workload of the different types of lectures, by other words, the number of hours of lectures per year and per week.

courseWorkload(courseID, type, totalQuantity, unitQuantity).
courseID: FK(course).

3.1.3.1 Single-Column Analysis

The results are summarized in the Table 3. The main causes behind null values in this table are courses without lectures (*e.g. Opção* (option in English)) and courses lectured in other universities (*e.g. Cibercrime - UL/FDireito*). The patterns in the *type* field are shown in Figure 5.

Table 3: Single-Column analysis for the table *courseWorkload*.

field	# of non-null values	# of unique values	# of null values	pattern (frequency)
courseID	3938	3938	0	
type	3938	7	360	
totalQuantity	3938	61	360	
unitQuantity	3938	61	360	

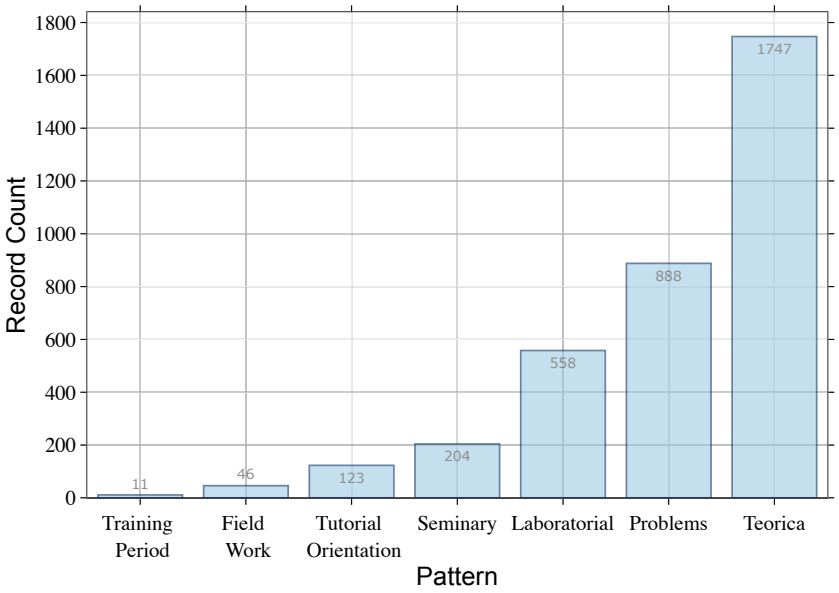


Figure 5: Pattern distribution for the field *type*.

3.1.4 Exam

This section shows the data profiling results for the table whose scheme is:

exam(id, startEnrolmentDate, endEnrolmentDate, startDate, endDate, type, isEnrolmentPeriod, courseID).
courseID: FK(course).

3.1.4.1 Single-Column Analysis

The results are summarized in the Table 4. The date format in this table is dd/mm/yyyy hh:mm. Furthermore, note that the number of null values in the fields *startEnrolmentDate* and *endEnrolmentDate* can be explained by the fact that these two fields are not mandatory.

Table 4: Record count for the fields of the table *exam*.

field	# of non-null values	# of unique values	# of null values	pattern (frequency)
id	3026	2836	1601	
startEnrolmentDate	3004	1017	1623	
endEnrolmentDate	3004	1000	1623	
startDate	3026	797	1601	
endDate	3026	839	1601	
isEnrolmentPeriod	3026	2	1601	true (81) false (2945)
type	3026	2	1601	Exam (173) Test (1295)
courseID	4627	2552	1601	

Table 5: Single-Column Analysis for the table *lecture*.

field	# of non-null values	# of unique values	# of null values	pattern (frequency)
name	37183	2741	593	
startDate	37183	1597	604	
endDate	37183	1704	604	
maximumNumberStudents	37183	160	593	
currentNumberStudents	37183	133	593	
type	37183	7	593	see Figure 6
roomName	37183	239	1851	
roomID	37183	239	1851	
courseID	37183	1449	0	

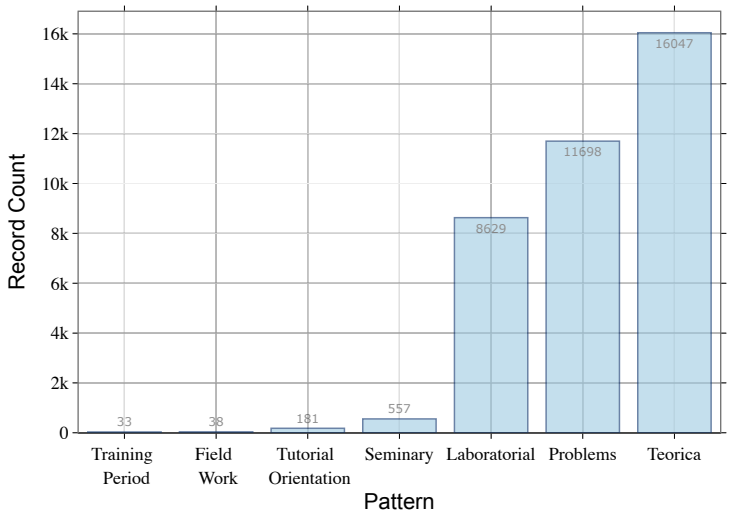


Figure 6: The patterns of the field type.

3.1.5 Exam Room

This table was not profiled as it is a subset of the table space.

3.1.6 Lecture

This section summarizes the data profiling results for the table whose scheme is:

lecture(name,startDate, endDate, maximumNumberStudents, currentNumberStudents, type, roomName, roomID, courseID).
roomID: FK(space).
courseID: FK(course).

The field *maximumNumberStudents* represents the maximum capacity that each lecture can support. The field *currentNumberStudents* represents the number of students currently enrolled in the lecture.

3.1.6.1 Single-Column Analysis

The results are summarized in the Table 5. The time format in this table is `yyyy-mm-dd hh:mm:ss`. Furthermore, the *ss* field, representing the seconds is never used. The pattern distribution of the field *type* is shown in Figure 6.

3.1.6.2 Multi-Column Analysis

3,8% of the records of the *Lecture* table have the value of the field *currentNumberStudents* with a value superior to the field *maximumNumberStudents* value. Table 6 shows examples of the lectures for which the number of enrolled students exceeds the maximum capacity.

Table 6: Example of lectures where the number of students enrolled exceeds the maximum capacity.

courseID	*currentNumberStudent*	*maximumNumberStudent*
283085589460982	36	35
283085589461024	6	0
1690460473010360	22	20
1690460473010470	28	23
1690460473010470	28	26
1690460473010480	30	23
1690460473010520	41	38
1690460473010530	1	0
1690460473010530	46	45
1690460473010530	56	55
1690460473010540	124	123
1690460473010560	19	18
1690460473010570	156	155
1690460473010590	30	29
1690460473010610	59	40
1690460473010610	21	19
1690460473010630	41	38
1690460473010730	15	0
1690460473010740	17	0

3.1.7 Course Degree

This section summarizes the data profiling results for the table whose scheme is:

courseDegree(idDegree,id).
idDegree: FK(degree).
id: FK(course).

This table is used to create a relationship between the courses and the degrees.

3.1.7.1 Single-Column Analysis

The results are summarized in the Table 7. All courses belong to a specific degree and therefore no null values should exist. However, sometimes the *idDegree* is empty in the *course* table. This incomplete information occurs when a course is lectured in different universities or a course is "fake", for example "OPÇÃO" (option in English).

Table 7: Single-Column analysis for the table *courseDegree*.

field	**# of non-null values**	**# of unique values**	**# of null values**	**pattern (frequency)**
idDegree	4451	86	360	
id	4811	4811	0	

3.1.8 Bibliographic References

This section summarizes the data profiling results for the table whose scheme is:

bibliographicReferences(id, author, year, title, reference, url, type, courseID).
courseID: FK(course).

3.1.8.1 Single-Column Analysis

The results are summarized in the Table 8.

Table 8: Single-Column Analysis for the table *bibliographicReferences*.

field	# of non-null values	# of unique values	# of null values	pattern (frequency)
id	13301	13301	0	
author	12623	5516	678	
year	12535	300	766	
title	12677	12677	624	
reference	12502	12502	799	
url	238	239	13063	
type	12677	2	624	MAIN (8788) SECONDARY (3889)
courseID	13301	1535	0	

Table 9: Single-Column Analysis for the table *space*.

field	# of non-null values	# of unique values	# of null values	pattern (frequency)
id	5383	5383	0	
name	5383	2173	1132	"??" (3) "A preencher pelo Gestor" (54)
capacity	4729	75	654	
capacityForExam	4729	39	654	
type	5383	5	0	see Figure 8
description	5383	2173	1132	"??" (3) "A preencher pelo Gestor" (54)
idParent	5380	221	3	Only the "CAMPUS" type has this value null

3.1.9 Space

This section summarizes the data profiling results for the table whose scheme is:

space(id, name, capacity, capacityForExam, type, description, idParent).
idParent: FK(space).

3.1.9.1 Single-Column Analysis

The results are summarized in Table 9. Figure 7 shows the different capacities from the different rooms. There are many rooms with a capacity smaller than 6. However, these rooms are not useful in the automatic timetabling system since their values are below the minimum number of students enrolled in a lecture.

3.1.9.2 Multi-Column Analysis

In this section, we compare the similar fields in order to detect redundant data and to validate the correctness of domain rules.

capacity and capacityForExam The *capacityForExam* should be less than or equal to the *capacity* for lectures. However, the data does not always hold this integrity constraint. Table 10 shows the rooms that have superior value in the field *capacityForExam* than in the field *capacity*.

name and description The field *name* and *description* have always the same data.

idParent and id The values that *idParent* takes always exist in the field *id*. *idParent* should be foreign key of table *space*.

Table 10: Rooms where normal capacity is smaller than in exam

name	*exam*	*normal*
C01	84	80
Lab Pedro Nunes	36	10

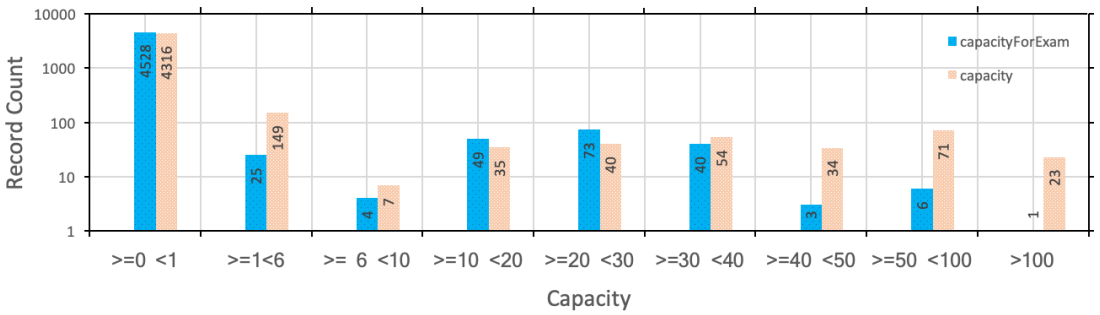


Figure 7: The room capacity at IST for exams and classes using a logarithmic scale.

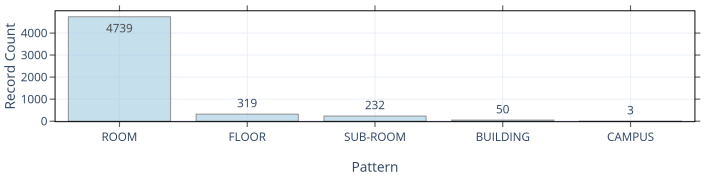


Figure 8: Distribution of values for the field for *type*.

3.1.10 Teacher

This section summarizes the data profiling results for the table whose scheme is:

teacher(id, name, courseID).
courseID: FK(course).

The goal of this table is to relate the course with the teacher that lectures it.

3.1.10.1 Single-Column Analysis

The results are summarized in the Table 11. The results show that the key for this table must be composed of two fields. The null values exists since some courses do not have the field id as they are "fake" courses.

3.1.11 Events by Room

This section summarizes the data profiling results for the table whose scheme is:

event(startDate, roomID, title, description, endDate, day, startHour, endHour, weekday).
roomID: FK(space).

In this section, the results are presented for the data available for the week of 5 June of 2017 since the API only provides information for the next week. There is no possible way of obtained events for more than one week via the current API. The standard data format for events is dd/mm/yyyy hh:mm.

Table 11: Single-Column analysis for the *teacher*.

field	# of non-null values	# of unique values	# of null values	pattern (frequency)
id	*2838*	*912*	*967*	
name	*2838*	*912*	*967*	
idcourse	*3473*	*1534*	*332*	

Table 12: Single-Column Analysis for the table *events*.

field	# of non-null values	# of unique values	# of null values	pattern (frequency)
startDate	4862	1667	0	0:00 (3196)
roomID	4862	4047	0	
title	4848	1416	14	
description	3455	117	1407	
endDate	4862	1667	0	0:00 (3196)
day	4862	7	0	
startHour	4862	97	0	
endHour	4862	108	0	
weekday	4862	7	0	

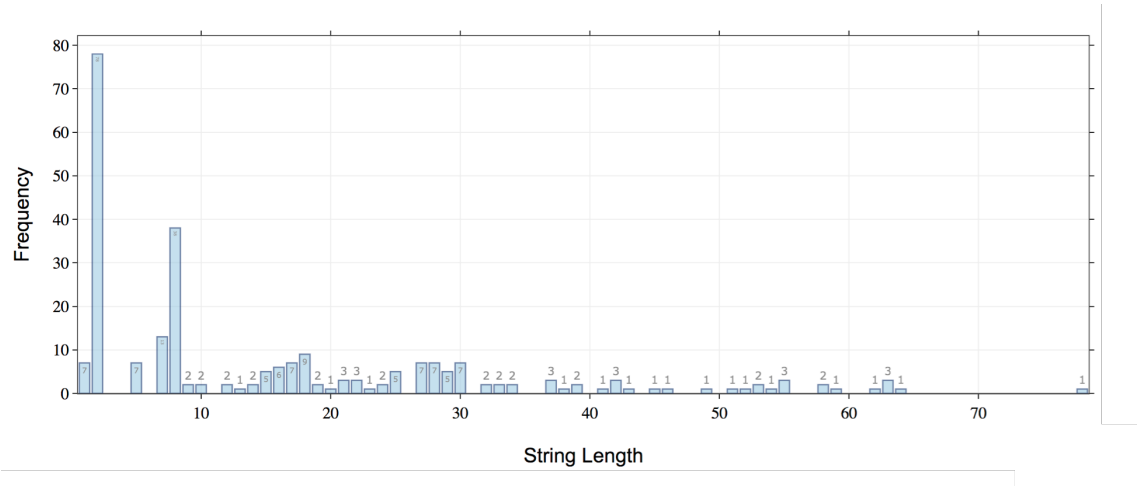


Figure 9: The string length frequency for the description field.

3.1.11.1 Single-Column Analysis

The results are summarized in Table 12. The important observations of these results are shown below.

title The number of null values in this field can be easily explained by the fact that tests and exams do not have this field.

description Only Generic events had null values in this column (*i.e.* when the event is "Test" or "Exam" the column is always non-null). The number of records with null values should be higher since some records have only a dot or a blank. This represents that the user did not want to introduce a title to the event. In fact, strings with sizes 1 and 2 have a frequency of 2.7% and 30.1% respectively. Figure 9 shows the string length frequency for the field.

3.1.11.2 Multi-Column Analysis

In this section, we compare the different date fields in order to detect redundant data and to validate the correctness of domain rules (*e.g.* an event must have a length large than zero).

StartDate and EndDate dates vs Day, StartHour and EndHour *StartDate* is always the same as *Day+StartHour* and *EndDate* is always the same as *Day+EndHour*.

Start and End dates We found that some events were scheduled with zero length and ends before they start. Among those events, only 7 would end before starting and 1408 had the same starting and ending hour (zero length). Table 14 shows the occurrences of events with starting hour $>$ ending hour. Table 15 shows an example of the occurrences of events with length 0.

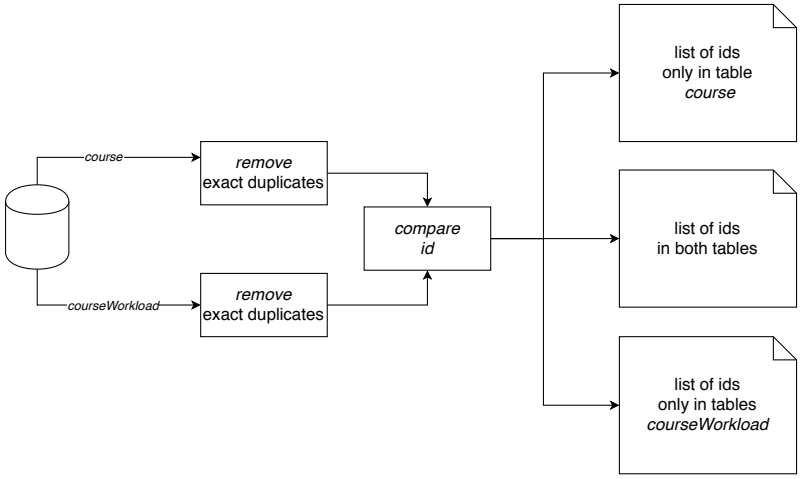


Figure 10: Procedure to compare the course id from two tables (*course* and *courseWorkloads*).

3.1.12 Inclusion Dependencies

After profiling data for each table we also checked the relationships between tables. From the statistical analysis performed before, we are aware of the primary keys and integrity constraint of each table. The goal of this section is to describe how we analyzed inclusion dependencies. Mainly, we want to validate the candidate foreign keys and to find redundant data scattered in the database.

To check the existence of a relationship between the different keys in the different tables, we must compare the existing values. With this goal in mind, it is possible to define a procedure to perform the comparison. This procedure can be applied to all the ids from the different tables. For example, Figure 10 shows the steps to compare the course ids from the table *course* and *courseWorkloads*. First, the exact duplicates records are removed since we only want to find if one *id* exists in the other table. The *compare id* operation compares the id fields from both tables and exports three lists of ids: the list containing the ids present in both tables, the list with the ids that only exist in the table *course*, and finally the list with the ids that only exist in the table *courseWorkloads*.

3.1.12.1 Degree and courseDegree

All id values in the *courseDegree* table exist in the *degree* table (and vice versa).

3.1.12.2 Course and courseDegree

All id values in the *courseDegree* table exist in the *course* table (and vice versa).

3.1.12.3 Course and courseWorkloads

All ids in the *courseWorkloads* table are present in the table *course*. However, not all course are present since no schedule is available. There are 15 courses without workload.

3.1.12.4 Events by Room and Space

All room id values in the *events* table exist in the *space* table. Naturally, the opposite does not happen.

3.1.12.5 Space and examRoom

All room ids in the *examRoom* table exist in the *space* table. Naturally, the opposite does not happen.

The data about the rooms (*name*, *capacity*, *capacityForExam*, *type*) stored in the table *examRoom* is an exact match of the data in the table *space* (fields with the same name).

3.1.12.6 Space and Lecture

All non-null ids on the *lecture* table exist on the *space* table. Naturally the opposite does not happen. The *roomName* available on the *lecture* table is always a match for the field *name* in table *space*. The

Table 13: Example of the problem where the number of students enrolled in lectures exceeds the capacity of the rooms. Empty spaces are rooms without capacity.

courseID	*name*	*currentC*	*normal*	*roomID*
1690460473010950	AAut25179T01	113	110	2448131364218
1690460473010450	AC25179PB06	43	38	2448131362951
1690460473009960	AFA9T01	14	0	2448131361392
1690460473010860	AID225179L04	19	18	2448131363680
1690460473010860	AID225179T01	43	0	2448131364297
1690460473010450	AC25179T01	122	118	2448131362979
1690460473010450	AC25179T02	123	118	2448131362979
1690460473010920	ACED1235179PB05	65	42	2448131362450
1690460473010920	ACED1235179T01	192	164	2448131363245
1690460473010870	ACED1245179PB03	58	54	2448131362947
1690460473010870	ACED1245179T01	127	90	2448131361607
1690460473010870	ACED1325179T01	148	130	2448131362371
1690460473010880	ACED1425179T01	143	130	2448131362372
1690460473010880	ACED1625179PB06	45	44	2448131362439
1690460473010880	ACED1625179T01	137	130	2448131362372
1690460473011010	ACED21179PB06	48	40	2448131365132
1690460473010840	ACED425179T01	224	115	2448131361602
1690460473011210	AE-IT01	74	60	2448131361732
1690460473010450	Aer25179T01	103	100	2448131362333
1690460473010790	VRui25179T02	127	94	2448131364033

Table 14: Events that end before they start.

Start	*End*	*roomID*	*title*
11/06/2017 09:00	11/06/2017 00:30	2448131363673	SALA DE ESTUDO
05/06/2017 09:00	05/06/2017 00:30	2448131363673	SALA DE ESTUDO
06/06/2017 09:00	06/06/2017 00:30	2448131363673	SALA DE ESTUDO
07/06/2017 09:00	07/06/2017 00:30	2448131363673	SALA DE ESTUDO
08/06/2017 09:00	08/06/2017 00:30	2448131363673	SALA DE ESTUDO
09/06/2017 09:00	09/06/2017 00:30	2448131363673	SALA DE ESTUDO
10/06/2017 09:00	10/06/2017 00:30	2448131363673	SALA DE ESTUDO

number of students enrolled in lectures sometimes exceeds the capacity of the rooms. Some examples are shown in Table 13.

3.2 Summary of Data Quality Problems Detected

The data profiling task identified the following data quality problems:

- Exact duplicates - there are 6,63% exact duplicates ids in the *course* table.
- Approximate Duplicates - there are 9,9% approximate duplicates in the field *name* of the *course* table. However, only 2% of the values in the field *name* are actually duplicates.
- Redundant data:
  - In the table *degree*, the fields *type* and *typename* have always the same value.
  - In the table *event*, the concatenation of the *startHour* (*endHour*) with *day* has always the same value as the field *startDate* (*endDate*).
  - Table *examRoom* is a subset of the table *space*. Table *examRoom* has 5,7% of the content of the table *space*
  - In the table *space* the fields *description* and *name* have always the same value for the type 'ROOM'. The fields have the same value in 79% of the entries.

Table 15: Events that start and end at the same time.

Start	*End*	*roomID*	*title*
05/06/2017 00:00	05/06/2017 00:00	2448131361538	Luis Rafael
05/06/2017 00:00	05/06/2017 00:00	2448131361544	Isabel Padrela
05/06/2017 00:00	05/06/2017 00:00	2448131361545	Rita Oom Temudo de Castro
05/06/2017 00:00	05/06/2017 00:00	2448131361546	Maria Amélia Alves Fernandes da Costa
05/06/2017 00:00	05/06/2017 00:00	2448131363780	Maria Jose Correia Martins Matias
05/06/2017 00:00	05/06/2017 00:00	2448131363895	José Maria Campos da Silva André
05/06/2017 00:00	05/06/2017 00:00	2448131364274	Mário Manuel Gonçalves da Costa
05/06/2017 00:00	05/06/2017 00:00	2448131364275	Luís Filipe Roriz
05/06/2017 00:00	05/06/2017 00:00	2448131364276	Toste Azevedo
05/06/2017 00:00	05/06/2017 00:00	2448131364279	Pedro Jorge Martins Coelho

- Percentage of null values in the whole database: 13,68%.
- Heterogeneous date representations: dd/mm/yyyy hh:mm (for the table *event* and *exam*), `yyyy-mm-dd hh:mm:ss` (*lecture*). The dd/mm/yyyy hh:mm format corresponds only to 11% of all dates in the database. Note that the *seconds* field always takes the value 0.
- Domain violation - room capacity with 0 to 6 (94,4%).
- Integrity constraint violation:
  - Exam capacity of room is bigger than its normal capacity (0,04% of the entries in the *space* table);
  - Start time of an event is after or the same as the ending time (15% of the entries in the table *event*);
  - The maximum capacity of a lecture is smaller than the number of enrolled students (3,8%);
  - The number of students in a lecture is bigger than the capacity of the room (3,8%).

Taking into account these data profiling results, the data cleaning recommendations are as follows:

1. It is important to remove the duplicates found and the tuples that do not satisfy integrity constraints.
2. Removing approximated duplicates is complicated but it can be solved by analysing multiple fields of the table.
3. In what concerns to incomplete data it can be just ignored as it has no impact on the timetables.
4. Convert the different date representation to a standard format.
5. Remove redundant data.

3.3 Comparison of the Data Profiling Tools used

There are many different data profiling tools in the market [1, 2]. In this work, we used PDI and Arrahtec's OSQD for profiling the data. We chose OSQD first since it is able to compute a long list of statistical based information and this could give a first glance of the data. This tool also supports analysis of some functional dependencies (*e.g.* checking if column A is a subset of column B). Furthermore, the tool automatically creates graphics for the different statistical analyses performed. However, it did not allow to perform a complex analysis, where the combinations of different fields were required to check some functional dependencies.

The major drawbacks of Arrahtec's OSQD are the following:

- The documentation provided by OSQD tool[5] is confusing.

[5]The documentation for Arrahtec's Open Source Data Quality and Profiling (OSQD) is available at https://arrahtech.github.io/docs/profiler_user_guide.

- The detection of approximate duplicates for data in the relational format does not return the expected result (*e.g.* it does not match two strings with an added letter difference (plural)).
- Some techniques for approximate duplicate detection are only available when comparing files. These methods were not tested as it would require to convert the tables to files (*e.g.* csv) since the JSON format is not supported.
- The implementation of the inclusion dependencies detection between two columns is hard to understand and does not show always what was expected.
- It is difficult to export the results (the tool automatically creates graphics). However, it is easy to export the statistics into csv.
- Complex multi-column analyses is impossible (*e.g.* checking if the room capacity is enough for the scheduled lecture).

Another example of a multi-column operation, that is impossible to perform in OSQD, is checking if the starting time of an event is after the ending time of the same event. Checking if the concatenation of two fields is always the same as another field is also impossible to do.

To perform the data profiling tasks, finally we chose PDI (without the DataCleaner plug-in) since this tool supported the missing functionality. PDI[6] has a large and comprehensive documentation with examples of usage. PDI allows the personalization of the different base operation available. This enables the execution of more complex data profiling operations (*e.g.* checking if the concatenation of two fields is always the same as another field). PDI was used to detect approximate duplicates and to perform multi-column analyses.

4 Data Transformation and Cleaning

This section describes the process of data cleaning and transformation. As mentioned in Section 1 the final relational model for the cleaned database is shown in Appendix A. The new database contains only the data useful to solve the timetable problem. The only difference between the diagram shown in Appendix A and the original diagram (Figure 2) is a reduction on the number of attributes represented. The cleaning process was execute using Cleenex, which will be described briefly in Section 4.1. The full cleaning programs and their cleaning graphs are shown in Appendix B.

4.1 Cleenex

Cleenex [5, 6] is an extensible data cleaning framework where the cleaning process is modeled as a data cleaning graph. Cleenex focuses on cleaning data from relational databases. Currently, Cleenex requires the data to be stored in PostGreSQL [13] relational database management system. The data cleaning graph is a direct acyclic graph where the nodes are data transformations and the edges are tables given as input or returned as the output of those transformations. Cleenex has a graphical user interface which enables the easy manipulation of the cleaning process by changing the cleaning graph or the code directly (via a text editor). Cleenex provides 5 logical operators: *view, match, map, merge and cluster*. The *view* operator encapsulates an arbitrary SQL query. The *match* operator accepts two tables as inputs and matches tuples based on an external distance/similarity function. The *map* operator iterates over each input record and allows the user to invoke external functions to transform the input attributes. The *cluster* operator executes a clustering algorithm, joining together those input tuples that are related according to a clustering algorithm. The *merge* operator merges partitions of the input data using a user-defined aggregation function. Some operators can invoke external functions that can be added to a pre-defined functions library.

Cleenex adds the possibility of involving the user in the cleaning process through Quality Constraints (QCs) and Manual Data Repairs (MDRs) [7]. These functionalities help the user to tune the data cleaning process and to manually correct entries that are impossible to be automatically corrected. QCs can be defined for each relation in the graph establishing a constraint which the data records must satisfy. MDRs allow the user to manually modify some records. MDRs are particularly useful to solve false positives and they will be used in this case study.

[6]PDI: Pentaho Data Integration documentation is available at http://community.pentaho.com/.

4.2 Applying Cleenex to transform and clean IST data

This section explains the data cleaning process using the prototype Cleenex. The transformation and cleaning process is discussed for each data quality problem in each table. For the complete data cleaning Cleenex programs see Appendix B.

4.2.1 Degree

This section presents the Cleenex data cleaning program to clean the data stored in the table with the following scheme:

degree(id, name, academicTerm, acronym, type, typename, objectives, description, designFor, links, operationRegime, gratuity, history, requisites, campusID).
campusID: FK(space).

There are no null values in this table. However, the data contained in it has redundancy, in particular, the columns *type* and *typename* have always the same value. Furthermore, the table contains data for degrees that are not currently in use and so they should be removed. We need to apply a data cleaning transformation whose resulting table has the *id* as primary key and fields *name*, *campusID* and *typename* (as type is a keyword in Cleenex) because these fields are the only ones useful for timetabling.

To correct the above-mentioned data quality problems, the Cleenex program, illustrated in Listing 2, can be used, where the *view* operator selects only the tuples with an *academicterm* of 2017/2018.

Listing 2: Cleenex data cleaning program to clean the Degree table.

```
CREATE VIEW cleanDegree
FROM degree d
WHERE d.academicterm="2017/2018"
{
SELECT d.name as name, d.id as id, d.campusid as campusid, d.typename as typename
KEY id
}
```

4.2.2 Teacher

This section presents the Cleenex data cleaning program to clean the data stored in the table with the following scheme:

teacher(id, name, courseID).
courseID: FK(course).

The only data quality problem present is the existence of incomplete data. This can easily be corrected by applying the *view* operator. The cleaning program is illustrated in Appendix B.2.

4.2.3 Space

This section presents the Cleenex data cleaning program to clean the data stored in the table with the following scheme:

space(id, name, capacity, capacityForExam, type, description, idParent).
idParent: FK(space).

There are two main concerns with the data from this table: (i) incorrect data in the *capacity* column; and (ii) room with correct *capacity* value but they are not classrooms.

It is impossible to solve incorrect data in the *capacity* column automatically. To correct them one requires user-input. This was achieved by the *mdr* shown in Listing 3.

Listing 3: Manual Data Repair for the incorrect capacities.

```
cleanSpace{
  UpdateCapacity: update (capacity,capacityForExam) using
```

```
  select *
  from cleanSpace
  where capacityForExam>capacity
  as view
}
```

The *mdr* allows the user to correct the capacity from table space when the field *capacityForExam* is larger than the field *capacity*.

To remove all elements that are not classrooms, it is important to know that classrooms and offices always have the *name* column with a non-null value. Other types of rooms (such as corridors, bathrooms, etc) can be removed by analyzing *name* column since these rooms do not have non-null values. However, floors, buildings, and campus must be kept, in order to know the location of each classroom. The main difference between offices and classrooms is the capacity. Offices have smaller capacities (bellow 6) and all classrooms have larger capacities. Using the *view* operator with the following *where* clause: name is not null AND capacity > 6 OR type $<>$ "ROOM" all the problems of the table are resolved. Rooms without capacities (such as corridors, bathrooms, etc) are removed by the null condition and the offices by the capacity. We ensure these clauses only applies to rooms. This way we can obtain the location of each classroom. The cleaning program is illustrated in Appendix B.3

4.2.4 Course Workload

This section presents the Cleenex data cleaning program to clean the data stored in the table with the following scheme:

courseLoad(courseID, type, totalQuantity, unitQuantity).
courseID: FK(course).

The only data quality problem found was the presence of incomplete data in the fields *type*, *totalQuantity* and *unitQuantity*. The record with incomplete data was removed by applying the *view* operator. The cleaning program is illustrated in Appendix B.4.

4.2.5 Course

This section presents the Cleenex data cleaning program to clean the data stored in the table with the following scheme:

course(id, name, academicTerm, acronym, numberStudents, evaluationMethod, program, curriculumID).

The main problem of this table is the existence of duplicates (both approximated and exact) for the field *name*. There are two types of approximate duplicates:

- The same name with the addition of punctuation marker (*e.g.* "."). For example, *"Mecânica Estatística e Transição de Fase"* (matched with *"Mecânica Estatística e Transição de Fase."*);
- A typographical error. For example, *"Introdução à Investigação em Engenharia Electrotécnica e de Computadores"* (matched with *"Introdução à Investigação em Engenharia Electrotécnica e de Computadore"*).

To clean approximate duplicates, it is necessary to choose which values are correct, and how to clean the ones that are incorrect. Therefore, one must know exactly which cases the data has. We applied Cleenex to help identify the different approximate duplicates.

In order to identify the different approximate duplicates one needs two steps: (i) a *matching* operator (Listing 4); and (ii) *mdr* to show the different matches (shown in Listing 5). To perform the first step we apply the Levenshtein distance. To reduce the number of tuples the user must manually check, a *view* operation is performed using the keyword *distinct* to remove exact duplicates.

Listing 4: Cleenex program to match approximate duplicates in *course* table.

```
CREATE MATCHING similarCourse
FROM courseWithID c, courseWithID t
```

```
LET sim = levenshteinDistance(c.name, t.name)
WHERE sim<=1 AND c.id<t.id {
SELECT t.id as id1,t.name as name1, c.id as id2, c.name as name2
}
```

One can use the *mdr* to correct the incorrect tuples found. However, in this case, it is not mandatory since two types of approximate duplicates found can automatically be corrected. We can take advantage of the unique *id* field. Therefore, it is easier to only perform an educated merge between the tuples with the same id.

The first type of approximate duplicates found can be corrected by applying a standardization procedure to the strings in question which can remove the problem of the inconsistent usage of punctuation markers and spaces (see Appendix B.5). The standardization removes spaces at the beginning and end of the string, as well as other incorrect characters in the course *name*.

Listing 5: Manual Data Repair for the similar courses.

```
similarCourse {
checkName: update (name1, name2) using
  select *
  FROM similarCourse
  as view
 }
```

The second type of approximate duplicates found can be corrected by always choosing the String with a larger number of characters. One can use this simple rule since all typographical errors reduced the size of the String. The merging function is shown in Appendix B.5.

Listing 6 shows the final Cleenex program to clean all the approximate duplicates. First, we apply the standardization procedure (removePredefinedPunctuation) and then we merge the courses based on the *id* field. Note that, the *merge* operator requires a field called *clusterId* to work. Therefore, we first rename the field *id* (line 6).

Listing 6: Final code to clean the table course.

```
CREATE MAPPING courseWithID
FROM course c
LET name=removePredefinedPunctuation(c.name){
SELECT name,c.academicterm as academicterm,
c.numberStudents as numberStudents,
c.id as clusterId, c.idCurriculum as idCurriculum
KEY clusterId}

CREATE MERGING cleanCourse
FROM courseWithID c
LET A = merge(c.clusterId, c.name,c.academicterm,c.numberStudents, c.idCurriculum)
{
SELECT A.name as name, A.id as id, A.term as academicterm, A.students as numberStudents,
    A.idc as idCurriculum
KEY id
}
```

4.2.6 Event by Room

This section presents the Cleenex data cleaning program to clean the data stored in the table with the following scheme:

event(startDate, roomID, title, description, endDate, day, startHour, endHour, weekday).
roomID: FK(space).

The data contained in this table had two data quality problems: (i) redundant data and (ii) incorrect data.

Recall that the fields *startDate* and *endDate* can be obtained by the concatenation of the fields *day* with *startHour* and *endHour*, respectively. Furthermore, there are tuples with incorrect data, where the *startDate* of an event is after or the same as the *endDate*. To clean this table the *view* operator is used. The code is shown in Listing 7.

Listing 7: Cleenex data cleaning program to clean table *event*.

```
1  CREATE VIEW cleanEvent
2  FROM event b
3  WHERE b.startcomplete <> b.endcomplete AND b.startcomplete<b.endcomplete
4  {
5  SELECT DISTINCT b.title as title,
6  b.info as info,b.roomid as roomid,
7  b.startcomplete as startcomplete,
8  b.endcomplete as endcomplete
9  KEY roomid,startcomplete
10 }
```

4.2.7 Lecture

This section presents the Cleenex data cleaning program to clean the data stored in the table with the following scheme:

lecture(name,startDate, endDate, maximumNumberStudents, currentNumberStudents, type, roomName, roomID, courseID).
roomID: FK(space).
courseID: FK(course).

This table has the following data quality problems: integrity constraint violations (*e.g. currentNumberStudents* > maximumNumberStudents), incomplete data (all fields have missing values except *courseID*) and exact duplicates (in the *name* field). Furthermore, there is an important data transformation task to be performed in this table: to obtain the periodicity of each lesson. To this end, one must merge all lectures of a course which are taught in the same weekday, hour and room but in different weeks.

The integrity constraints in this table relate to the capacity of lectures, to the capacity of rooms and to the number of students enrolled. These constraints only represent ideal goals, and therefore they are not mandatory. For this reason, the tuples that violate the integrity constraints are not a problem when considering the generation of timetables. They provide useful information about the quality of the timetable.

The cleaning process is shown in Listing 8. The cleaning process is separated in the following steps: (i) remove incomplete data (*view* operation in lines 1-9); (ii) create two new fields by separating the day from the time contained in the *startDate* and *endDate* fields (lines 11-20); (iii) compute the weekdays based on the new *day* field (lines 22-31); (iv) match two lectures taught in the same weekday, time and room (lines 33-45); (v) cluster all matched lectures (lines 47-50); and finally (vi) merge all the lectures of the same cluster (lines 52-59).

Listing 8: Cleenex code to clean table *lecture*.

```
1  //Step (i)
2  CREATE VIEW lectureNotNull
3  FROM lecture t
4  WHERE t.nameshifft IS NOT NULL AND t.endDate IS NOT NULL AND t.room IS NOT NULL
5  {
6  SELECT t.typeT as typeT, t.course as id, t.nameshifft as name,
7  t.room as roomid,t.currentC as current, t.maxC as maxC,t.endDate as endDate,
8  t.start as startDate KEY name,startDate,roomid
9  }
10
11 //Step (ii)
12 CREATE MAPPING lectureTime
```

```
13 FROM lectureNotNull t
14 LET endTime=splitDateTime(t.endDate),startTime=splitDateTime(t.startDate),
15 day=splitDate(t.endDate)
16 {
17 SELECT t.typeT as typeT, t.id as id, t.name as name,
18 t.roomid as roomid,t.current as current, t.maxC as maxC, endTime, startTime,day
19 KEY name,startTime,day,roomid
20 }
21
22 //Step (iii)
23 CREATE MAPPING lectureDay
24 FROM lectureTime t
25 LET weekday=dayOfWeek(t.day)
26 {
27 SELECT t.typeT as typeT, t.id as id, t.name as name,
28 t.roomid as roomid,t.current as current, t.maxC as maxC, t.endTime as endTime,
29 t.startTime as startTime,t.day as day,weekday
30 KEY name,startTime,day,roomid
31 }
32
33 //Step (iv)
34 CREATE VIEW lectureMatcheed
35 FROM lectureDay c, lectureDay t
36 WHERE t.name=c.name AND t.weekday=c.weekday AND t.startTime=c.startTime AND
       t.roomid=c.roomid
37 {
38 SELECT t.typeT as typeT1, t.id as id1, t.name as name1,
39 t.roomid as roomid1,t.current as current1, t.maxC as maxC1,
40 t.endTime as endTime1, t.startTime as startTime1,t.day as day1,
41 t.weekday as weekday1,c.typeT as typeT2, c.id as id2, c.name as name2,
42 c.roomid as roomid2,c.current as current2, c.maxC as maxC2, c.endTime as endTime2,
43 c.startTime as startTime2,c.day as day2,c.weekday as weekday2
44 KEY name1,startTime1,day1,roomid1,name2,startTime2,day2,roomid2
45 }
46
47 //Step (v)
48 CREATE CLUSTERING clusterLectures
49 FROM lectureMatcheed r
50 BY TransitiveClosure
51
52 //Step (vi)
53 CREATE MERGING cleanLecture
54 FROM clusterLectures c
55 LET A = merge(c.clusterId, c.typeT1,c.id1,c.roomid1,
       c.current1,c.maxC1,c.endTime1,c.startTime1,c.day1,c.weekday1)
56 {
57 SELECT A.name as name, A.id as id, A.roomID as roomid, A.currentC as currentC, A.maxC as
       maxC,A.endTime as endTime, A.startTime as startTime, A.day as day,A.weekday as
       weekday, A. periodStart as periodStart, A.periodEnd as periodEnd
58 KEY name, startTime, day
59 }
```

Some steps require the addition of Java functions to perform specific needs of the data business model, which are explained below. Step (ii) requires the creation of functions to manipulate dates. The functions *splitDateTime* and *splitDate* return the time (hh::mm:ss.000000) and date (yyyy-MM-dd) contained in date (yyyy-MM-dd hh::mm:ss.000000), respectively. Step (iii) requires a similar function (*dayOfWeek*) to return weekday based on a date (yyyy-MM-dd). The Java code for these functions is shown in Appendix B.7.3. Step (v) uses the Transitive Closure method for the clustering. Step (vi) requires a function that receives, as input, the fields corresponding to all the clustered lectures and returns a newly merged lecture (line 49). The only field with different values within the clustered lectures is *day* (with the information corresponding to the day of the month). Therefore, this field is the only one which is complex to merge. The merged lecture has two new fields: *periodStart* and

periodEnd. These fields correspond to first and last date (yyyy-MM-dd) where the lecture is taught. The corresponding Java code is shown in Appendix B.7.1. and B.7.2.

4.2.8 Exam and Exam Room

This section presents the Cleenex data cleaning program to clean the data stored in the table with the following scheme:

exam(id, startEnrolmentDate, endEnrolmentDate, startDate, endDate, type, isEnrolmentPeriod, courseID).
courseID: FK(course).

examRoom(examID, roomID, capacity, capacityForExam, type, name).
roomID: FK(space).
examID: FK(exam).

The table *exam* has incomplete data in all the fields (with particular focus on the fields corresponding to dates) which needs to be removed. Furthermore, the data about enrolments (*startEnrolmentDate*, *endEnrolmentDate* and *isEnrolmentPeriod*) contained in the table is not useful to timetabling problems. Therefore, these data should be removed from table *exam*. The cleaning program is illustrated in Appendix B.8.

The table *examRoom* is a subset of the table *space* and therefore contains redundant data. Originally the idea was to merge the table *examRoom* with the table *exam*. However, this is not possible since there are exams without rooms associated. To this end, the table exam room continues to exist but with less data (only the *examID* and the *roomID*). The cleaning program is illustrated in Appendix B.9.

To sum up, the problems of both tables can be solved by applying the *view* operator.

4.2.9 Bibliographic References

This section presents the Cleenex data cleaning program to clean the data stored in the table with the following scheme:

bibliographicReferences(id, author, year, title, reference, url, type, courseID).
courseID: FK(course).

The data contained in *Bibliographic References* table has no influence over the generation of timetables. Therefore, it can be removed.

4.2.10 Course Degree

This section presents the Cleenex data cleaning program to clean the data stored in the table with the following scheme:

courseDegree(idDegree,id.)
idDegree: FK(degree).
id: FK(course).

The data quality problems found in *courseDegree* have no influence on the generation of timetables since the courses without degrees do not have actual classes to be scheduled. Therefore, all fake courses can just be removed. The cleaning program is illustrated in Appendix B.10.

4.3 Suggestions for Cleenex Expansion

After using Cleenex to clean the IST academic data some ideas to improve its usability are summarized below:

- Data types: types like bigint/text/time exist in PostgreSQL relational database system but not in Cleenex. As Cleenex does not support all data types, some data types were considered Strings (*e.g.* time). The identifiers are also converted to strings as they were larger than the type integer.

- Show the errors on the website console instead of the server console.
- List of keywords that cannot be used (*e.g. type*). Could be highlighted by the code editor.
- Synchronization between the code and the graph. Sometimes the changes made to the code do not automatically translate into the graph and vice-versa. In these cases, it is required to open and close the program to update the graphical user interface.
- Synchronization between the front-end and back-end. Sometimes when one changes the name of a table in the code, the result can be seen in the graph but both the logs and the Postgres database continue to have the old name.
- Short cuts. For example, "Ctrl+s" to save the changes in the work.
- Define the project name when saving.
- Add a horizontal scroll on the code editor.
- Bug-fix: Removing an object from the graphical interface does not always delete everything on the code ($\{$ is not removed).
- Add support priorities for multiple conditions in the *where* clause. When considering the condition a AND b OR c it should be possible to define which Boolean function is more important. This can be addressed by allowing the usage of '()' to prioritize expressions.
- Support case sensitive code (now everything is converted to lower-case).
- Bug-fix: Resizing the console window does not resize correctly the code window.
- Store the current position of the graph when saving. This way the cleaning graph is not "hidden" when one reopens the project.
- Bug-fix: Merge operation requires the input table to have an argument called cluster Id. It should allow the use of the key of the table (without renaming the field).
- Bug-fix: Clustering operation has a limit number of fields that receives as input for the clustering procedure. This may imply new operations to have all data clustered.

5 Conclusion

Profiling the data obtained from the Fénix system showed several data quality problems that range from redundant data to integrity constraint violations. In terms of data profiling tools, PDI is more complex to use than OSQD but it has also more functionality. In terms of statistics, OSQD provides a wide range of possibilities. We applied OSQD to obtain the statistics for single-column analysis and PDI to detect inclusion and functional dependencies.

For data cleaning, the Cleenex system was used. The system was able to clean all data quality problems. However, due to the lack of types available, some fields were considered as Strings (but this does not affect the quality of the solution). The cleaned data is stored in a relational database whose structure was organized accordingly to the necessary data to automate the generation of timetables.

Acknowledgements

This work was supported by national funds through Fundação para a Ciência e a Tecnologia (FCT) with reference SFRH/BD/143212/2019 (PhD grant) and UIDB/50021/2020 (INESC-ID multi-annual funding).

References

[1] Abedjan, Z., Golab, L., Naumann, F.: Profiling relational data: a survey. The VLDB Journal 24(4), 557–581 (2015)

[2] Barateiro, J., Galhardas, H.: A survey of data quality tools. Datenbank-Spektrum 14(15-21), 48 (2005)

[3] Bonutti, A., Cesco, F.D., Gaspero, L.D., Schaerf, A.: Benchmarking curriculum-based course timetabling: formulations, data formats, instances, validation, visualization, and results. Annals Operations Research 194(1), 59–70 (2012)

[4] Fernandes, P., Pereira, C.S., Barbosa, A.: A decision support approach to automatic timetabling in higher education institutions. Journal of Scheduling 19(3), 335–348 (2016)

[5] Galhardas, H.: Nettoyage de Données Déclaratif: Langage, Modèle, et Algoritmes. Ph.D. thesis, Université de Versailles Saint-Quentin-en-Yvelines (2001)

[6] Galhardas, H., Florescu, D., Shasha, D.E., Simon, E., Saita, C.: Declarative data cleaning: Language, model, and algorithms. In: International Conference on Very Large Data Bases (VLDB), Roma, Italy. pp. 371–380 (2001)

[7] Galhardas, H., Lopes, A., Santos, E.: Support for user involvement in data cleaning. In: International Conference on Data Warehousing and Knowledge Discovery (DaWaK), Toulouse, France. pp. 136–151 (2011)

[8] Ham, K.: Openrefine (version 2.5). http://openrefine. org. free, open-source tool for cleaning and transforming data. Journal of the Medical Library Association: JMLA 101(3), 233 (2013)

[9] Lemos, A., Melo, F.S., Monteiro, P.T., Lynce, I.: Room usage optimization in timetabling: A case study at Universidade de Lisboa. Operations Research Perspectives 6, 100092 (2019)

[10] Lemos, A., Monteiro, P.T., Lynce, I.: Minimal perturbation in university timetabling with maximum satisfiability. In: Proceedings of 16th International Conference on Integration of Constraint Programming, Artificial Intelligence, and Operations Research (CPAIOR)

[11] Levenshtein, V.I.: Binary codes capable of correcting deletions, insertions, and reversals. In: Soviet physics doklady. vol. 10, pp. 707–710 (1966)

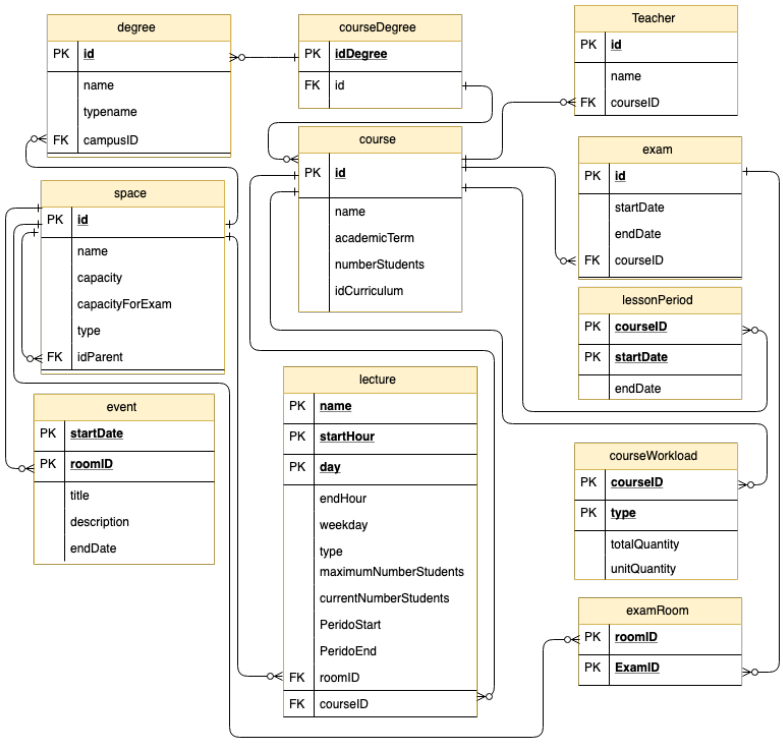
[12] Schaerf, A.: A survey of automated timetabling. Artificial Intelligence Review 13(2), 87–127 (1999)

[13] Stonebraker, M., Rowe, L.A.: The design of postgres. In: ACM SIGMOD International Conference on Management of Data, Washington, DC, USA. pp. 340–355 (1986)

[14] de Werra, D.: An introduction to timetabling. European Journal of Operational Research 19(2), 151 – 162 (1985)

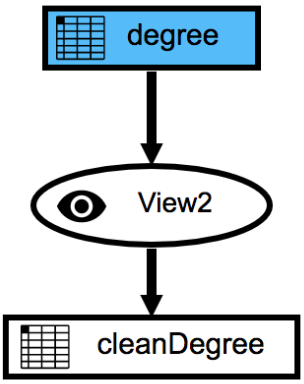
Appendices

A Relational model of the cleaned database



B Data Cleaning: The Complete Programs

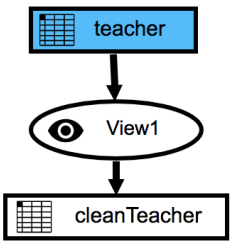
B.1 Degree




```
1  DEFINE CONSTANTS AS
2  DEFINE COMPOSED TYPES AS
3  DEFINE FUNCTIONS AS
4  DEFINE ALGORITHMS AS
5  TransitiveClosure
6  DEFINE INPUT DATA FLOWS AS
7
8  TABLE degree
9  (
10     name  STRING(100),
```

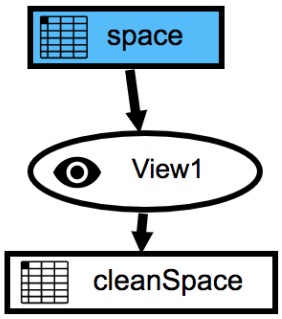
```
    id STRING(26),
    academicterm STRING(21),
    acronym STRING(15),
    typeT STRING(30),
    typename STRING(30),
    campusid STRING(26),
    description STRING(2000),
    obj STRING(2000),
    design STRING(2000),
    requisites STRING(2000),
    history STRING(2000),
    operationregime STRING(2000),
    gratuity STRING(2000),
    links STRING(2000)
);

DEFINE TRANSFORMATIONS AS

CREATE VIEW cleanDegree
FROM degree d
WHERE d.academicterm="2017/2018"
{
SELECT d.name as name, d.id as id, d.campusid as campusid, d.typename as typename
KEY id
}
```

B.2 Teacher




```
DEFINE CONSTANTS AS
DEFINE COMPOSED TYPES AS
DEFINE FUNCTIONS AS
DEFINE ALGORITHMS AS
TransitiveClosure
DEFINE INPUT DATA FLOWS AS

TABLE teacher
(
    name STRING(60),
    id STRING(26),
    idcourse STRING(26)
);

DEFINE TRANSFORMATIONS AS

CREATE VIEW cleanTeacher
FROM teacher t
WHERE t.name IS NOT NULL AND t.id IS NOT NULL AND t.idcourse IS NOT NULL
{
SELECT t.name as name, t.id as id, t.idcourse as idcourse
KEY id, idcourse
}
```

B.3 Space

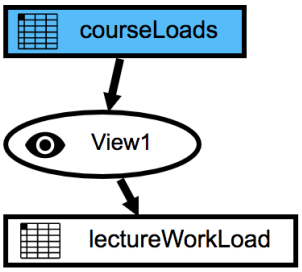



```
DEFINE CONSTANTS AS
DEFINE COMPOSED TYPES AS
DEFINE FUNCTIONS AS
DEFINE ALGORITHMS AS
TransitiveClosure
DEFINE INPUT DATA FLOWS AS

TABLE space
(
    name STRING(200),
    id  STRING(25),
    idparent STRING(25),
    typet STRING(50),
    exam integer,
    normal integer,
    description STRING(200)
);

DEFINE TRANSFORMATIONS AS

CREATE VIEW cleanSpace
FROM space s
WHERE s.name is not null AND s.normal >6 OR s.typet<>"ROOM"{
SELECT s.normal,s.exam, s.id, s.idparent,s.name,s.typet
}
```

MDR

```
cleanSpace{
  UpdateCapacity: update (capacity,capacityForExam) using
  select *
  from cleanSpace
  where capacityForExam>capacity
  as view
}
```

B.4 Course Workload




```
DEFINE CONSTANTS AS
DEFINE COMPOSED TYPES AS
DEFINE FUNCTIONS AS
DEFINE ALGORITHMS AS
TransitiveClosure
DEFINE INPUT DATA FLOWS AS

TABLE courseLoads(typeT STRING(50),unit int, total int, course STRING(26));

DEFINE TRANSFORMATIONS AS

CREATE VIEW lectureWorkLoad
FROM courseLoads t
WHERE t.typeT IS NOT NULL
{
SELECT t.typeT as typeT, t.course as id, t.unit as unit, t.total as total
KEY typeT, id
}
```

B.5 Course

B.5.1 First approach

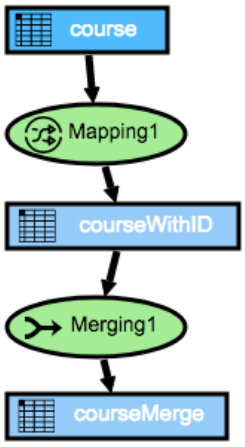
```

DEFINE CONSTANTS AS
DEFINE COMPOSED TYPES AS
DEFINE FUNCTIONS AS
GenerateID.generateId(INTEGER) RETURN STRING
Levenshtein.levenshteinDistance(STRING, STRING) RETURN INTEGER
MergeName.mergeSimple(OBJECT[], OBJECT[]) RETURN STRING
Convert.removePredefinedPunctuation(STRING) RETURN STRING
DEFINE ALGORITHMS AS
TransitiveClosure
DEFINE INPUT DATA FLOWS AS

TABLE course
(
    name STRING(140),
    academicterm STRING(70),
    acronym STRING(15),
    numberofattendingstudents integer,
    evaluationmethod STRING(400),
    program STRING(400),  id STRING(26),
    idc STRING(26));

DEFINE TRANSFORMATIONS AS
CREATE VIEW courseWithoutExactDuplicates
```

```
FROM course c
WHERE c.idc is not null
{
SELECT DISTINCT c.name as name,c.academicterm as academicterm,
c.numberofattendingstudents as numberofattendingstudents,
c.id as id, c.idc as idc
KEY name
}


CREATE MATCHING similarCourse
FROM courseWithoutExactDuplicates c, courseWithoutExactDuplicates t
LET sim = levenshteinDistance(c.name, t.name)
WHERE sim<=1 AND c.idg<t.idg {
SELECT t.id as id1,t.name as name1, c.id as id2, c.name as name2
}
```

MDR

```
similarCourse {
ckeckName: update(name1,name2) using
  select *
  FROM similarCourse
  as view
 }
```

B.5.2 Final approach




```
DEFINE CONSTANTS AS
DEFINE COMPOSED TYPES AS
Course
(
    name STRING(600),
    term STRING(70),
    students INTEGER,
    id STRING(26),
    idc STRING(26));
DEFINE FUNCTIONS AS
GenerateID.generateId(INTEGER) RETURN STRING
MergeName.merge(OBJECT[], OBJECT[]) RETURN Course
Convert.removePredefinedPunctuation(STRING) RETURN STRING(400)
DEFINE ALGORITHMS AS
TransitiveClosure
DEFINE INPUT DATA FLOWS AS
```

```
TABLE course
(
    name STRING(140),
    academicterm STRING(70),
    acronym STRING(15),
    numberStudents INTEGER,
    evaluationmethod STRING(400),
    program STRING(400),  id STRING(26),
    idCurriculum STRING(26));

DEFINE TRANSFORMATIONS AS
CREATE MAPPING courseWithID
FROM course c
LET name=removePredefinedPunctuation(c.name){
SELECT name,c.academicterm as academicterm,
c.numberStudentsnumberStudents as numberStudents,
c.id as clusterId, c.idCurriculum as idCurriculum
KEY clusterId}

CREATE MERGING cleanCourse
FROM courseWithID c
LET A = merge(c.clusterId, c.name,c.academicterm,c.numberStudents, c.idCurriculum){
SELECT A.name as name, A.id as id, A.term as academicterm, A.students as numberStudents,
    A.idc as idCurriculum
KEY id
}
```

Merge Function

```
@ExportFunction
   public static Course merge(Object[] id, Object[] name, Object[] term,Object[]
       students, Object[] idc){
      Course c= new Course();
      String result=(String)name[0];
      int place=0;
      for(int i=0; i<name.length;i++){
         result=(((String)name[i]).length()>result.length()?(String)name[i]:result);
         place=(((String)name[i]).length()>result.length()?i:place);
}
      c.id=(String)id[place];
      c.idc=(String)idc[place];
      c.students=(Integer)students[place];
      c.name=(String)result;
      c.term=(String)term[place];

      return c;
   }
```

Data Structure

```
public class Course {
   public String id;
   public String idc;
   public String term;
   public String name;
   public int students;
}
```

B.5.3 Standardization Function

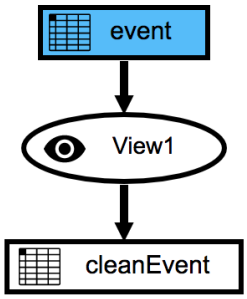
```
@ExportFunction
```

```
  public static String removePredefinedPunctuation(String str) {

     return removePunctuation(str, "!@$%&/(){}\"\"\'\'-_,:.;?=+*").trim();
  }
```

This function which was already defined in Cleenex, was also used:

```
@ExportFunction
 public static String removePunctuation(String str, String punct) {
       StringTokenizer st = new StringTokenizer(str,punct);
       StringBuffer sb = new StringBuffer();
       if (st.hasMoreElements()) sb.append(st.nextToken());
       while (st.hasMoreElements())
           sb.append(" " + st.nextToken());
       return sb.toString();
    }
```

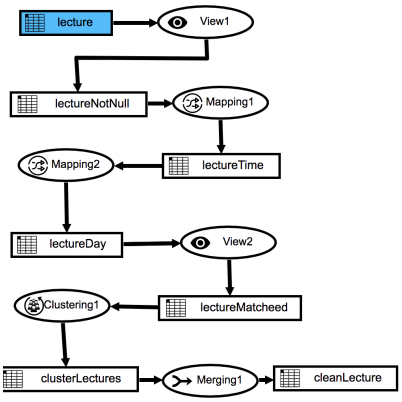
B.6 Events by Room




```
DEFINE CONSTANTS AS
DEFINE COMPOSED TYPES AS
DEFINE FUNCTIONS AS
DEFINE ALGORITHMS AS
TransitiveClosure
DEFINE INPUT DATA FLOWS AS

TABLE event
(
    roomid STRING(25),
    typeT STRING(10),
    startHour STRING(6),
    endHour STRING(6),
    startDate STRING(36),
    endDate STRING(36),
    weekday STRING(4),
    day date,
    info STRING(200),
    title STRING(200)
);

DEFINE TRANSFORMATIONS AS

CREATE VIEW cleanEvent
FROM event b
WHERE  b.startcomplete <> b.endcomplete AND b.startcomplete<b.endcomplete
{
SELECT DISTINCT b.title as title,
b.info as info,b.roomid as roomid,
```

```
b.startcomplete as startcomplete,
b.endcomplete as endcomplete
KEY roomid,startcomplete,title
}
```

B.7 Lecture




```
DEFINE CONSTANTS AS
DEFINE COMPOSED TYPES AS
lecture (
roomID STRING(26),
name STRING(70),
typeT STRING(100),
currentC INTEGER,
maxC INTEGER,
startTime STRING(15),
endTime STRING(15),
day STRING(10),
periodStart STRING(10),
periodEnd STRING(10),
weekday STRING(10),
id STRING(26));

DEFINE FUNCTIONS AS
GenerateID.generateId(INTEGER) RETURN STRING
MergeName.merge( OBJECT[],OBJECT[], OBJECT[],OBJECT[], OBJECT[],OBJECT[],
    OBJECT[],OBJECT[], OBJECT[], OBJECT[]) RETURN lecture
DateFunction.splitDateTime(STRING) RETURN STRING(15)
DateFunction.splitDate(STRING) RETURN STRING(10)
DateFunction.dayOfWeek(STRING) RETURN STRING(10)

DEFINE ALGORITHMS AS
TransitiveClosure

DEFINE INPUT DATA FLOWS AS
TABLE lecture
(
typeT STRING(30),
nameshifft STRING(30),
currentC int,
maxC int,
start STRING(40),
endDate STRING(40),
course STRING(26),
room STRING(40),
nameR STRING(70)
);
```

```
40
41
42  DEFINE TRANSFORMATIONS AS
43
44  CREATE VIEW lectureNotNull
45  FROM lecture t
46  WHERE t.nameshifft IS NOT NULL AND t.endDate IS NOT NULL AND t.room IS NOT NULL
47  {
48  SELECT t.typeT as typeT, t.course as id, t.nameshifft as name,
49  t.room as roomid,t.currentC as current, t.maxC as maxC,t.endDate as endDate,
50  t.start as startDate KEY name,startDate,roomid
51  }
52
53  CREATE MAPPING lectureTime
54  FROM lectureNotNull t
55  LET endTime=splitDateTime(t.endDate),startTime=splitDateTime(t.startDate),
56  day=splitDate(t.endDate)
57  {
58  SELECT t.typeT as typeT, t.id as id, t.name as name,
59  t.roomid as roomid,t.current as current, t.maxC as maxC, endTime, startTime,day
60  KEY name,startTime,day,roomid
61  }
62
63  CREATE MAPPING lectureDay
64  FROM lectureTime t
65  LET weekday=dayOfWeek(t.day)
66  {
67  SELECT t.typeT as typeT, t.id as id, t.name as name,
68  t.roomid as roomid,t.current as current, t.maxC as maxC, t.endTime as endTime,
69  t.startTime as startTime,t.day as day,weekday
70  KEY name,startTime,day,roomid
71  }
72  CREATE VIEW lectureMatcheed
73  FROM lectureDay c, lectureDay t
74  WHERE t.name=c.name AND t.weekday=c.weekday AND t.startTime=c.startTime AND
        t.roomid=c.roomid
75  {
76  SELECT t.typeT as typeT1, t.id as id1, t.name as name1,
77  t.roomid as roomid1,t.current as current1, t.maxC as maxC1,
78  t.endTime as endTime1, t.startTime as startTime1,t.day as day1,
79  t.weekday as weekday1,c.typeT as typeT2, c.id as id2, c.name as name2,
80  c.roomid as roomid2,c.current as current2, c.maxC as maxC2, c.endTime as endTime2,
81  c.startTime as startTime2,c.day as day2,c.weekday as weekday2
82  KEY name1,startTime1,day1,roomid1,name2,startTime2,day2,roomid2
83  }
84
85  CREATE CLUSTERING clusterLectures
86  FROM lectureMatcheed r
87  BY TransitiveClosure
88
89
90  CREATE MERGING cleanLecture
91  FROM clusterLectures c
92  LET A = merge(c.clusterId, c.typeT1,c.id1,c.roomid1,
        c.current1,c.maxC1,c.endTime1,c.startTime1,c.day1,c.weekday1)
93  {
94  SELECT A.name as name, A.id as id, A.roomID as roomid, A.currentC as currentC, A.maxC as
        maxC,A.endTime as endTime, A.startTime as startTime, A.day as day,A.weekday as
        weekday, A. periodStart as periodStart, A.periodEnd as periodEnd
95  KEY name, startTime, day
96  }
```

B.7.1 Merge Function for Lectures

```
1    public static lecture merge(Object[] clusterId, Object[] roomid, Object[] typeT,
         Object[] id, Object[] current, Object[] maxC, Object[] endTime, Object[]
         startTime, Object[] day, Object[] weekday){
2         lecture ret = new lecture();
3       ret.roomID = (String) roomid[0];
4       ret.name = (String) clusterId[0];
5       ret.day = (String) day[0];
6       ret.typeT = (String) typeT[0];
7       ret.currentC = (int) current[0];
8       ret.maxC = (int) maxC[0];
9       ret.startTime = (String) startTime[0];
10      ret.endTime = (String) endTime[0];
11      ret.id = (String) id[0];
12      ret.weekday = (String) weekday[0];
13      ret.periodStart= (String) day[0];
14      ret.periodEnd= (String) day[0];
15      for (int i = 0; i < clusterId.length; i++){
16
17            SimpleDateFormat dateformat=new SimpleDateFormat("yyyy-MM-dd");
18            Date date,date1,date2,date3;
19            try {
20               date = dateformat.parse((String)day[i]);
21
22               date2 = dateformat.parse((String)ret.periodStart);
23               date3 = dateformat.parse((String)ret.periodEnd);
24               if(date.before(date2))
25                  ret.periodStart=(String)day[i];
26
27               if(date.after(date3))
28                  ret.periodEnd=(String)day[i];
29
30            } catch (ParseException e) {
31                  //
32            }
33
34
35       }
36     return ret;
37
38    }
```

B.7.2 Data Structure

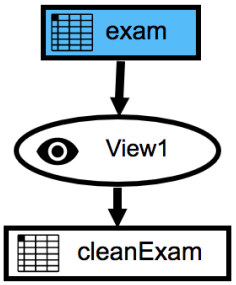
```
1  public class Lecture {
2
3     public String roomID;
4     public String name;
5     public String typeT;
6     public int currentC;
7     public int maxC;
8     public String startTime;
9      public String periodStart;
10     public String periodEnd;
11    public String endTime;
12    public String day;
13    public String weekday;
14    public String id;
15 }
```

B.7.3 Date Manipulation Functions

```
@ExportClass(path = "date")
public class DateFunction {

      /**
     *  Returns time from string date+time
     *  @param String containing the date in the follwing format yyyy-MM-dd hh:mm:ss.000000
     *  @return String with time hh:mm:ss.000000
     *  @exception
     **/
    @ExportFunction
    public static String splitDateTime(String dateStr){
       String date[] = dateStr.split(" ");
       return date[1];
    }
    /**
     *  Returns date from string date+time
     *  @param String containing the date in the follwing format yyyy-MM-dd hh:mm:ss.000000
     *  @return String with date yyyy-MM-dd
     *  @exception
     **/
    @ExportFunction
    public static String splitDate(String dateStr){
       String date[] = dateStr.split(" ");
       return date[0];
    }

    /**
      *  Returns the weekday of the date
      *  @param String containing the date in the follwing format
      *  @return weekday
      *  @exception
      **/
    @ExportFunction
    public static String dayOfWeek(String dateStr){
       SimpleDateFormat dateformat=new SimpleDateFormat("yyyy-MM-dd");
       Date date;
       try {
          date = dateformat.parse(dateStr);
          return dayOfWeek(date);
       } catch (ParseException e) {
          System.out.println(e.getMessage());
       }
       return "";
     }
    /**
     *  Returns the weekday of the date
     *  @param date in the following format
     *  @return weekday
     *  @exception
     **/
    @ExportFunction
    public static String dayOfWeek(Date date){
       DateFormat dayFormate=new SimpleDateFormat("EEEE");
       return dayFormate.format(date);
    }



     public static void main(String[] args) throws Exception {
       String input_date_string="2017-07-20";
```

```
        System.out.println("Day of the week for 2017-07-20 : " +
            dayOfWeek(input_date_string));

    }
}
```

B.8 Exam



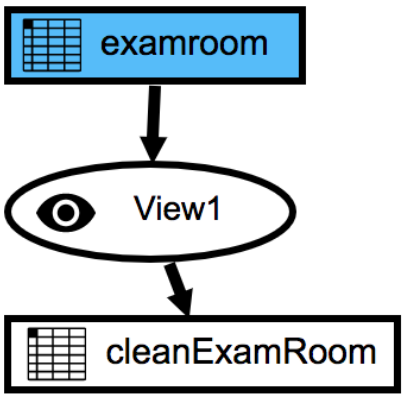

```
DEFINE CONSTANTS AS
DEFINE COMPOSED TYPES AS
DEFINE FUNCTIONS AS
DEFINE ALGORITHMS AS
TransitiveClosure
DEFINE INPUT DATA FLOWS AS

TABLE exam
(
    id STRING(26),
    name STRING(30),
    typeT STRING(30),
    isEnrolmentPeriod BOOL,
    startDate STRING(36),
    endDate STRING(36),
    startEnrolmentDate STRING(36),
    endEnrolmentDate STRING(36),
    idcourse STRING(26)
);

DEFINE TRANSFORMATIONS AS

CREATE VIEW cleanExam
FROM exam c
WHERE c.id IS NOT NULL
{
SELECT c.id as id, c.name as name,
c.typeT as typeT,c.evaluationperiodstart as evaluationperiodstart,
c.evaluationperiodend as evaluationperiodend, c.idcourse as idcourse
 KEY id
}
```

B.9 Exam Room

```
DEFINE CONSTANTS AS
DEFINE COMPOSED TYPES AS
DEFINE FUNCTIONS AS
DEFINE ALGORITHMS AS
TransitiveClosure
DEFINE INPUT DATA FLOWS AS
```

```

TABLE examroom
(
    examid STRING(26),
    name STRING(200),
    typeT STRING(50),
    id STRING(26),
    exam integer,
    normal integer,
    description STRING(200)
);

DEFINE TRANSFORMATIONS AS


CREATE VIEW cleanExamRoom
FROM examroom c
WHERE c.id IS NOT NULL
{
SELECT c.id as roomid, c.examid as id
KEY id,roomid
}
```

B.10 Course Degree




```
DEFINE CONSTANTS AS
DEFINE COMPOSED TYPES AS
DEFINE FUNCTIONS AS
DEFINE ALGORITHMS AS
TransitiveClosure
DEFINE INPUT DATA FLOWS AS

TABLE courseDegree
(
    idDegree STRING(26),
    id STRING(26)
);

DEFINE TRANSFORMATIONS AS


CREATE VIEW cleancourseDegree
FROM courseDegree c
WHERE c.idDegree IS NOT NULL
```

```
{
SELECT c.id as courseID, c.idDegree as idDegree
KEY courseID,idDegree
}
```
