Lavoisier: A DSL for increasing the level of abstraction of data selection and formatting in data mining

Alfonso de la Vega*, Diego García-Saiz, Marta Zorrilla, Pablo Sánchez

Software Engineering and Real-Time (ISTR), University of Cantabria
Avda. Los Castros s/n, 39005 Santander, Cantabria, Spain

Abstract

Input data of a data mining algorithm must conform to a very specific tabular format. Data scientists arrange data into that format by creating long and complex scripts, where different low-level operations are performed, and which can be a time-consuming and error-prone process. To alleviate this situation, we present *Lavoisier*, a declarative language for data selection and formatting in a data mining context. Using Lavoisier, script size for data preparation can be reduced by ∼40% on average, and by up to 80% in some cases. Additionally, accidental complexity present in state-of-the-art technologies is considerably mitigated.

Keywords: Data Selection, Data Formatting, Domain-Specific Languages, Data Mining

1. Introduction

We live in a time where data analysis techniques are becoming very popular, as they have demonstrated to be beneficial for the success of an organisation or project. Examples exist in multiple domains, such as agriculture [1], (bio)medical areas [2, 3], system security [4], or solid-state materials research [5]. Despite this extended usage, executing data mining processes still requires performing a lot of low-level technical tasks, where an explicit and fine-grained

*Corresponding author
Email addresses: alfonso.delavega@unican.es (Alfonso de la Vega), diego.garcia@unican.es (Diego García-Saiz), marta.zorrilla@unican.es (Marta Zorrilla), p.sanchez@unican.es (Pablo Sánchez)

management of multiple details is mandatory [6]. As a result, the level of abstraction at which data scientists typically work is very low, which hinders their productivity.

Among these technical tasks, the most time-consuming one is typically the selection and preparation of data for an analysis [7]. Most data mining algorithms, such as those found in the Weka [8], KNIME [9] or scikit-learn [10] libraries, require their input data to be arranged in a very specific two-dimensional tabular format, where all the information related to each entity under analysis must be placed in a single row. For example, if we were analysing businesses by using information about sales, business providers and customers satisfaction, all this information, for each business, would have to be placed into cells of a single row of the table providing input data. This means that these algorithms cannot work with hierarchical or linked data such as JSON or XML files, or relational tables connected by means of foreign keys, which are common examples of representations in which information is typically made available. Therefore, to execute a data mining algorithm, we first need to transform data stored in these representations into the specific tabular format that these algorithms can process.

Data scientists perform this data transformation process by creating long and complex scripts, written in data management languages such as SQL (*Structured Query Language*) [11], R [12], or Pandas [13] (i.e. a well-known Python data manipulation library). These scripts extract data from the available sources and, through a set of low-level operations, such as *joins* [14] or *pivots* [15, 16], arrange these data as a tabular dataset that fulfils the *one entity, one row* constraint previously commented. The elaboration of these scripts, which is a crucial step for the outcome of any analysis [17], can be a tedious, time-consuming and error-prone process.

To alleviate this situation, we present Lavoisier, a language that aims to automate some of the data management tasks that data scientists need to perform when building datasets. To automate these tasks, Lavoisier provides a set of declarative constructs that focus on specifying what information, among the

available in a certain domain, should be included in a concrete analysis. These constructs are automatically processed by the language interpreter through different chains of data transformation operations, such as joins and pivots. Therefore, using Lavoisier, data scientists can focus on specifying what data must be selected for a certain analysis, and forget about the details of how the selected data must be transformed to conform a dataset, which contributes to increase their productivity.

For instance, let A and B be entities of a domain, where instances of A can refer to several B instances through a *bs* relationship ($A \xrightarrow{bs} B$), and each B instance has a *b_id* identifier attribute. To generate a properly formatted dataset of A instances, including the information of *bs*, the expression in Lavoisier would be `mainclass A include bs by b_id`. On the contrary, to achieve the same result using SQL or Pandas, a 2-to-3 times longer and more complex expression would be required. Precisely, we would need to perform a left join between A and B, followed by a pivot. Moreover, some extra fine-grained operations to, for instance, avoid name collisions between attributes of the combined entities, might also be required. This scenario is detailed in Section 3.4, using concrete entities from a business reviews domain.

The expressiveness and effectiveness of Lavoisier were assessed by a comparison against the two technologies mentioned in the previous paragraph, which are currently very popular for data manipulation: SQL [11] and Pandas [13]. In the comparison, a comprehensive set of data selection and preparation scenarios were initially devised, using two different case studies for this purpose. Then, for each scenario, we compared the corresponding Lavoisier specification with its SQL and Pandas' counterparts. As a result of this comparison, we concluded that Lavoisier's dataset specifications are more compact, less verbose and allow working at a higher abstraction level. In general, script size can be reduced on average due to the use of Lavoisier by 60% and 40% with respect to SQL and Pandas, and by up to 80% in some cases. This script size reduction is mainly caused by Lavoisier's dataset specifications requiring 40% fewer operations on average, and 70% less parameters than SQL and Pandas' counterparts.

This paper updates and extends a previous contribution presented at the *9th International Conference on Model and Data Engineering (MEDI)* [18]. Over this contribution, we include:

- a more detailed context and problem statement in Section 2,
- a revised and extended description by example of the language in Section 3,
- a description of the implementation, which was not included in the conference version of this paper, and which presents the internal structure of Lavoisier in Section 4, and
- an extended and more rigorous evaluation of our work in Section 5, where more extraction scenarios and an additional case study have been included.

After the evaluation, Section 6 comments on related work and, finally, Section 7 summarises this article and outlines future work.

2. Case Study and Problem Statement

This section describes with more detail the motivation behind this work. To illustrate it, we use the *Yelp Dataset Challenge*, which is introduced next.

2.1. Running Example: The Yelp Dataset Challenges

Yelp is a company that provides an online business review service. Using this service, owners can describe and advertise their businesses and customers can write their opinions about these businesses. Yelp periodically collects and makes available bundles of data for academic usage in the form of data analysis challenges[1]. We will use these data and challenges as running example throughout this article.

Yelp provides their data as a bundle of interconnected JSON (*JavaScript Object Notation*) files. To help visualise these files, we inferred the conceptual

[1] `https://www.yelp.com/dataset/challenge`

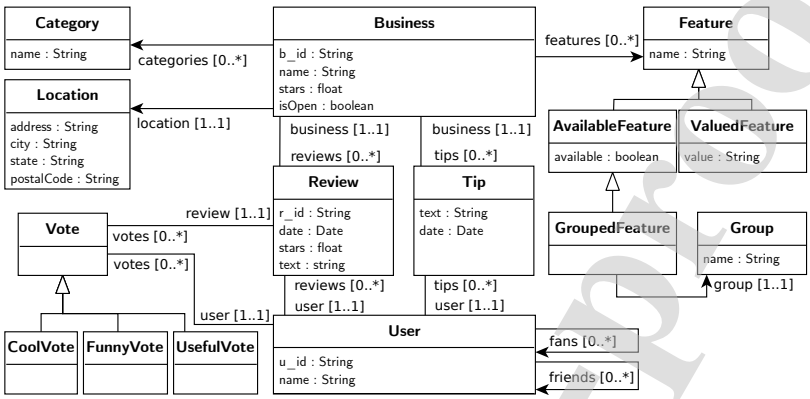

Figure 1: Conceptual Model of the Yelp Dataset Challenge.

object-oriented model to which these files would conform to. This model is depicted in Figure 1.

For each registered *business*, Yelp provides information about its *location*; the different *features* it offers, such as the availability of Wi-Fi or a smoking area; and the *categories* which best describes it, e.g., Cafes, Restaurant, Italian, Gluten-Free, and so on. *Users* can *review* these businesses, rate them and write a text describing their experience. Additionally, users can write *tips*, which are small pieces of advice about a business, such as *do not miss its salmon!* Yelp also provides some social network capabilities, so users can have *friends* or *fans*. Users can also receive *votes* in their reviews in case other users found these reviews *funny*, *useful* or *cool*.

Using these data, Yelp raises some questions to be solved by executing analysis tasks, such as identifying reasons behind a business becoming successful. In the following section, we explain how to execute data mining processes that aim to provide answers to these questions, detailing all the steps that need to be accomplished.

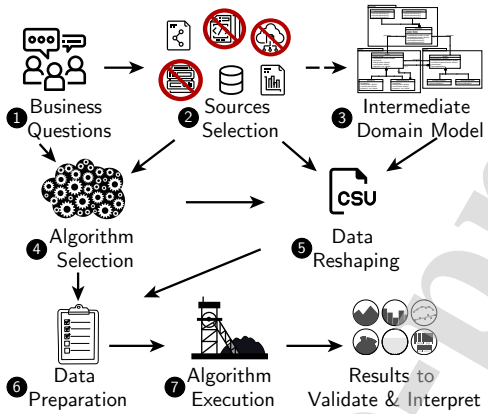

Figure 2: Detailed data mining process.

2.2. Data Mining Processes

To describe the steps that comprise a data mining process, we use as reference the *KDD (Knowledge Discovery in Databases)* process [19]. Some steps of this process have been split into several sub-stages, in order to clarify better some important issues related to the work described in this paper.

Any data mining process is created to answer business questions (Figure 2, Step 1). These questions are derived from the business or domain experts needs. For instance, in the case of Yelp, data mining processes are elaborated to try to answer the questions raised in their challenges, such as finding reasons that make a business successful.

Once the questions from the domain are identified, we need to decide what data sources will be used to answer them (Figure 2, Step 2). These sources are also selected with the help of the domain experts. We might use several data sources. For instance, Yelp challenges' data include information from different subsystems, such as the review system or its social network.

When the data to analyse are not trivial and have (complex) relationships between them, it might be worth to construct a *domain model* that abstracts from the low-level representation of these bundles and helps visualise and un-

derstand them [20] (Figure 2, Step 3). This domain model might be specifically helpful in the cases where several data sources are used, each one following a different data representation. In these cases, the domain model can be used to provide an unified view of the data available in the domain. As an additional benefit, the existence of such an abstracted view could help ease the participation of business experts in the definition of the mining process [21]. In the case of Yelp, they extracted data from different subsystems, unified them and provided to researchers as a set of interconnected JSON files. To improve understanding of these data, we abstracted these JSON files into the conceptual data model depicted in Figure 1.

Subsequently, we must select the data mining technique that we consider most adequate for answering each business question (Figure 2, Step 4). For instance, *clustering techniques* [6] might be employed to group Yelp businesses according to the similarity of their characteristics. This might give indications of what commonalities are shared among successful and unsuccessful businesses.

For each data mining technique, such as *clustering*, *association rules* or *classification*, there is a plethora of algorithms available in the literature. Each one of these algorithms is designed to perform better than the others depending on certain characteristics of the input data. Therefore, we are also in charge of selecting the algorithm that best fits with the nature of our input data. For instance, to analyse data from Yelp, a density-based clustering algorithm like DBSCAN [22] might perform better than a centroids-based one, such as k-means [23].

Once a data mining algorithm has been selected, we would need to feed this algorithm with input data. Most data mining algorithms can only accept as input data arranged in a very specific tabular format. Data scientists often refer to bundles of data arranged in this format as *datasets*. Therefore, as next step of a data mining process, we need to reshape the available data to create *datasets* that can be digested by data mining algorithms (Figure 2, Step 5). The work presented in this paper focuses on this specific step, which will be explained more in detail in the next section.

In addition to this reshaping, each algorithm might impose other extra constraints to their input data. For instance, some distance-based algorithms require data to be normalised into the range [0, 1]. Therefore, we would need to perform some extra data transformations in order to ensure these constraints are satisfied. These transformations can take place after a dataset has been produced or at the same time it is generated (Figure 2, Step 6).

Finally, we execute the selected data mining algorithms with the generated datasets (Figure 2, Step 7) and we obtain the output results. These results must be analysed to asses their quality and reliability. Then, curated results can be passed to the business or domain experts (Figure 2, Step 8), who would interpret them to extract some conclusions and make some decisions. As an example, after performing an analysis over data from a Yelp challenge, the obtained results might provide interesting advice for an entrepreneur before starting a new venture.

Next section details the stage of this data mining process in which this work focuses: the creation of tabular datasets from non-tabular information, such as linked or hierarchical data.

2.3. The Data Formatting Problem

As previously commented, data mining algorithms only accept input data in a specific tabular format, where all the information about each instance of the domain entity being analysed is placed in a single row. The problem of achieving this format is described in this section.

To illustrate this problem, let us suppose that, in the context of the Yelp challenge, we want to identify business features, or combinations of features, that might lead a business to have a high stars rating. We decide to use as information for the analysis the businesses id and name, so business can be easily identified; their stars rating as a measure of success; and their set of available features[2]. Figure 3 (a) shows the fragment of the original domain model that contains this

[2]The *Features* inheritance of Figure 1 has been omitted from this initial example.

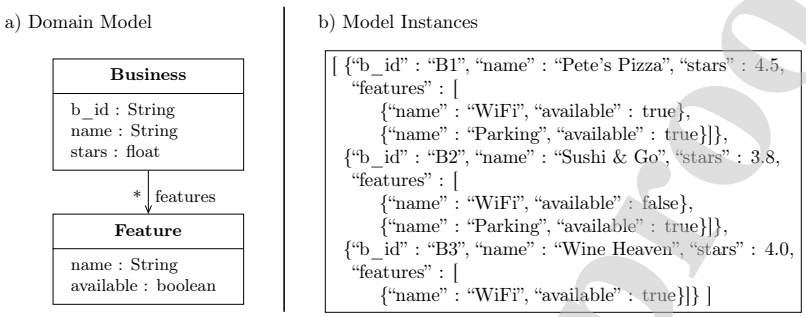

Figure 3: (a) Yelp model fragment; (b) JSON data conforming to (a).

B_id	BName	Stars	WiFi	Parking
B1	Pete's Pizza	4.5	true	true
B2	Sushi & Go	3.8	false	true
B3	Wine Heaven	4.0	true	

Figure 4: A tabular dataset of businesses' data.

information, whereas Figure 3 (b) shows some data, in JSON format, conforming to this model fragment. According to the model, each business might have several nested features, making this information hierarchical. Therefore, we would need to find a mechanism to convert the hierarchical information of each business into a flat vector of data that can be placed in single dataset row, such as depicted in Figure 4.

To address this formatting problem, data scientists currently rely on data management languages or libraries, such as SQL [11] or Pandas [13]. Using these technologies, data scientists create scripts that, by means of several operations, such as *filters*, *joins*, *pivots* or *aggregations*, are able to convert hierarchical and linked information into tabular datasets. In the following, we refer to these operations as *low-level* ones for the purpose of building datasets, because they can be used to select and transform data, but they are not specifically designed for producing tabular data that satisfies the *one row, one entity* constraint. Consequently, data scientists usually have to combine several of these operations

Listing 1: SQL flattening operation using aggregation queries and the case operator.

```
1  select b.b_id, b.name, b.stars,
2         max(case features.name when 'Parking'
3             then available end) as feature_parking,
4         max(case features.name when 'WiFi'
5             then available end) as feature_wifi,
6         max(case features.name when 'Smoking'
7             then available end) as feature_smoking
8  from business as b
9  left join features on b.b_id = features.b_id
10 group by b.b_id, b.name, b.stars
```

to ensure this constraint. Therefore, data scientists need to care both about *what* data they want to select, and *how* the selected data must be transformed to produce a dataset that can be digested by data mining algorithms. On the other hand, we refer to an operation as *high-level* when this operation allows data scientists to just focus on the selection of data, skipping over any details of how these data should be arranged in a tabular format to satisfy the *one row, one entity* constraint.

As an example, Listing 1 shows an SQL script where, by employing a *CASE* operator in an aggregation query, the tabular representation of Figure 4 can be obtained from a set of relational tables conforming to the domain model fragment of Figure 3 (a). At a first glimpse, this script is quite complex when compared to the data model size, that is composed of two small classes. Different operations are used, such as a *join* of the *features* and *business* tables in line 9, or a combination of aggregations with a *group by* expression to apply a reformatting operation similar to a *pivot* in lines 2-7 and 10 (more details on the pivot operation are given in Sections 3.4 and 5.2). In addition, it should be noticed that this SQL query would require an update if the features that businesses can offer change, since a new aggregation query needs to be included

each time a new feature is added to the system. This might be a problem if these values change very often, or when there is a high number of these values.

Some data management languages, such as Pandas [13], provide a more robust implementation of the pivot operator where business features would not need to be explicitly listed, as they would be automatically calculated. Nevertheless, we would still need to concatenate several operations to select and transform data, as again these languages provide just operations to handle data, but not specific or dedicated ones for selecting data and automatically arranging them into a tabular format satisfying the *one entity, one row* constraint. A detailed Pandas script for a data selection scenario can be found in Section 5.2.

Therefore, the operations provided by these languages, according to the given definitions at the start of this section, can be considered low-level, because they are used to select the data of interest (i.e. business and features information), and to explicitly transform the structure of the selected data to arrange them in the required format.

To alleviate this situation, we studied how to create a language to provide: (1) a set of *high-level* primitives to specify which data, among the available in a domain, should be included in a specific dataset; and (2) an interpreter that processes these primitives to automatically execute the low-level data transformation operations that are required to generate the desired dataset.

With such a language, data scientists would not have to create complex, time-consuming and error-prone data reformatting scripts by hand, which should contribute to increase their productivity. Therefore, in this context, the reasons behind defining this new language can be related to the *task automation* DSL decision pattern [24]. This is, we are creating a DSL for automating some data selection and transformation tasks. This language, which we called Lavoisier, is described in the next section.

3. Lavoisier: Dataset Extraction Language

Since Lavoisier aims to be a language for selecting a subset of all data available in a domain, we need a mechanism to describe these data. This mechanism should be a high-level notation, such as conceptual modelling notation, that allows us to focus on domain data and their relationships, and avoids technical issues about how these data are stored. From the different candidate notations, we selected object-oriented models, such as the one in Figure 1, since this technique is widely used nowadays to define domain models [25], and the use of domain models has been previously proposed as a way to relate domain knowledge with data mining and machine learning solutions [20]. Therefore, Lavoisier can be more rigorously defined as *a high-level language to create datasets from object-oriented models describing domain data.*

With respect to the type of primitives that are contained in the language, Lavoisier's syntax allows users to select *what* parts of an object-oriented model are to be processed and included in an output dataset. However, users do not need to worry about *how* those parts would be processed and merged into the appropriate dataset format, as that process is transparently performed by Lavoisier in the background.

To determine what primitives to include in Lavoisier, we analysed the different structural elements that might appear in an object-oriented model. Then, we studied how each one of these elements might be transformed into a tabular dataset according to different scenarios. We formalised each one of these transformation scenarios by means of a *flattening operator*. Finally, we created a set of primitives to process these scenarios, trying both to maintain a low number of total primitives in Lavoisier, and to make these as simple to use as possible by reducing the information that needs to be specified by the end user.

In the remaining of this section, we describe Lavoisier by example, showing how the language can deal with data transformation scenarios of increasing complexity. We focus on describing how a data scientist can select data from an object-oriented model using Lavoisier, while abstracting from the low-level

transformations that would be performed in each transformation scenario. However, as previously commented, each one of these transformations is formally specified by means of a *flattening operator*, which provides a more precise semantic definition of the Lavoisier primitives. Since the focus of this paper is to describe Lavoisier from a high-level point of view, and for the sake of brevity, we have left the definition of such an operator outside of the scope of this paper. A preliminary version on this operator can be found in [26], whereas a more complete and definitive version is provided in [27]. We refer the interested reader to these works.

3.1. Single Class Selection

An example of the most basic Lavoisier snippet that can be expressed is shown in Listing 2. In Lavoisier, data selection processes are performed by defining *dataset specifications*, which are declared with the *dataset* keyword followed by a *name* for the dataset to be created (for instance, *yelp_reviews* in Listing 2), and a *body* block surrounded by braces (lines 1-3). A dataset specification must always declare a *main class* (line 2), which is the class whose instances would be placed in the rows of the output dataset. In this case, the main class is *Review*.

Listing 2: Lavoisier's simplest dataset specification.

```
1 dataset yelp_reviews {
2   mainclass Review
3 }
```

By default, if we do not provide any further information, Lavoisier considers that all attributes of the selected class must be included in the output dataset, whereas references to other classes are excluded. Taking into account these considerations, Lavoisier generates a dataset from the specification of Listing 2 as follows: first, a table with a column for each attribute contained in the *Review* class is generated. Then, each instance of the *Review* class is processed, generating a new row where the instance attributes are assigned to their corresponding columns. Figure 5 shows an example of this output.

r_id	date	stars	text
5559	2020/02/06	4	It was a great ...
5560	2020/02/07	2	Not really ...
...	...	...	...

Figure 5: Output dataset example for Listing 2.

This default behaviour is not enough for most cases, since we might be interested in excluding attributes like *text*, or including other related information, such as the author of the review. Next sections describe how to carry out these and other actions.

3.2. Attributes and Instances Filtering

If we do not want to include all attributes of a class in the resulting dataset, we can specify those of our interest as a list after the class name, between brackets, such as illustrated in Listing 3. In this example, just the *r_id* and *stars* attributes would be selected. The resulting dataset is created following the process of the previous subsection, but this time only columns for those attributes specified in the list between brackets would be generated.

Listing 3: Limiting the list of extracted attributes to *r_id* and *stars*.

```
1 dataset yelp_reviews {
2   mainclass Review [r_id, stars]
3 }
```

Also, it might be the case that we are not interested in including all instances of a class in a particular dataset. For these cases, Lavoisier provides filters. A *filter* is specified after a class name using the *where* keyword, and it must contain a predicate that is evaluated for each class instance. If the predicate evaluates to *true*, then the instance is processed; otherwise it is discarded. If any attribute included in the predicate had *null* as value, the whole predicate evaluates to false. Listing 4 shows a filter example for considering just reviews belonging to businesses placed in Edinburgh (line 3). The current prototype of the language allows performing typical boolean operations such as equality

or inequality comparisons, and a combination of these with {*and*, *or*} boolean conjunctions. For future versions, we plan to include support for more sophisticated operations usually present in data processing languages, such as functions for managing dates or strings.

Listing 4: Selecting *Review* instances from Edinburgh businesses only.

```
1 dataset yelp_reviews {
2   mainclass Review [r_id, stars]
3       where business.location.city = "Edinburgh"
4 }
```

3.3. Single-Bounded Reference Inclusion

When analysing a main class, we might be also interested in including information of a secondary entity that can be accessed through a reference of that class. For instance, when analysing reviews, we might want to add information about the author of a review. In this case, author data can be retrieved by navigating the *user* reference of the *Review* class.

Incorporating references is more challenging than including attributes because of two main reasons: (1) referenced types might have their own attributes and nested references that, in turn, we might want to include or exclude; and (2) data of the referenced class must be merged with data of the main class in order to create a single table. The complexity of this data merging process depends on the cardinality of the included reference, with two cases being distinguishable: (1) references with 1 as upper bound, which we denote as *single-bounded references*; and (2) references with an upper bound greater than 1, to which we refer as *multi-bounded references*.

In Lavoisier, single-bounded references can be incorporated to a dataset with the *include* primitive. For example, in Listing 5, line 3, the *user* reference of the *Review* class is included in the dataset through the *include user* statement. As in the previous section, if no extra information is given, all attributes of the included class (e.g. *User* in this case) would be placed in the output dataset, and its references would be excluded.

r_id	date	stars	text	user_u_id	user_name
5559	2020/02/06	4	It was a great ...	7777	John
5560	2020/02/07	2	Not really ...	8998	Mary
...	...	...	...	...	...

Figure 6: Example Output Dataset for Listing 5

Listing 5: Reviews data with some attribute and reference selections.

```
1 dataset yelp_reviews {
2   mainclass Review
3   include user
4 }
```

The data merging process in the case of single-bounded references is rather simple. First, a new column is added to the output table per each simple attribute contained in the referenced class. In the case of Listing 5, the *u_id* and *name* attributes of the *User* class would be added to the output table. To avoid name collisions, the name of these new columns is formed by concatenating the reference name with the attribute being included. So, in the example, the columns would be formed by prepending the *user* reference name, resulting in *user_u_id* and *user_name*. When processing an instance of the main class, the associated instance of the referenced class is accessed, and their attributes placed in their corresponding columns. For example, when processing a specific review, its author would be retrieved by Lavoisier, and the values for their *u_id* and *name* attributes placed in the appropriate columns. Conceptually, this is equivalent to performing a *left outer join* between the main class and the referenced class, and then transforming the resulting class to a table.

3.4. Multi-Bounded Reference Inclusion

To illustrate the main problem when processing references with upper bounds greater than one, let us consider the example of Figure 3 (a). In this case, we want to analyse businesses and include in the analysis information about each business features. Since the *features* reference has upper bound greater than

(a) After join: business information ends in several rows.

B_id	BName	Stars	Feature	Available
B1	Pete's Pizza	4.5	WiFi	true
B1	Pete's Pizza	4.5	Parking	true
B2	Sushi & Go	3.8	WiFi	false
B2	Sushi & Go	3.8	Parking	true
B3	Wine Heaven	4.0	Wifi	true

(b) After join and pivot: Information of each business placed in a single row.

B_id	BName	Stars	WiFi_available	Parking_available
B1	Pete's Pizza	4.5	true	true
B2	Sushi & Go	3.8	false	true
B3	Wine Heaven	4.0	true	

Figure 7: Data merging by combining *join* and *pivot* operations.

one, each instance of a *Business* might be related with several instances of the *Feature* class. It should be noticed that, in addition, the set of features associated to a business might have a variable size. Therefore, we need to find a mechanism to transform these nested data with a variable size into fixed-length data vectors, so that these vectors can be used as rows for an output table defining a dataset. That is, for each business, we need to distribute the information about each one of their features over a well-defined set of columns. This implies that each potential business feature must have associated a set of columns where its information can be placed. Therefore, we would need to create a set of columns for the *WiFi* feature, so that when a business has *WiFi*, information about this feature can be placed in those columns.

To achieve this goal, a data scientist might chain two low-level operations: a *join* between both classes, and then, a *pivot*. These operations are described in the following, with the help of Figure 7, and using as example the classes and data of Figure 3.

The first operation, the *join*, is a typical *left outer join* that produces as output a table like illustrated in Figure 7 (a). After performing this join, the

resulting table contains several rows referring to the same business, which violates the *one entity, one row* constraint. For instance, the first two rows refer to the *Pete's Pizza* business, showing the availability of the *WiFi* and *Parking* features. However, the information of these two rows should be contained in a single one. To solve this issue, a data scientist might execute a pivot. A *pivot* is an operation that accepts three parameters: a set of *static properties*, a set of *pivoting properties*, and a set of *pivoted properties*. To achieve our desired result, in our case, the static properties always correspond to all the properties of the main class; the pivoting properties are the set of properties of the referenced class that can identify their instances; and, as pivoted properties, the remaining properties of the referenced class are used. For the concrete example of Figure 7, we use as pivoting property the *name* of the feature, so *available* becomes the pivoted property.

With these parameters, the pivot would work as follows in this case. Firstly, the structure of the output table is determined. To do it, all static properties, which would be *B_id*, *BName* and *Stars* in our example, are added as columns to the output table. Then, all distinct tuples of the pivoting properties are calculated. In the example, these tuples would be (*WiFi*) and (*Parking*). They represent and identify the set of all potential features that might be associated to a business. Then, each one of these tuples is combined with the pivoted properties to conform the new set of columns that should be created. In our case, the *WiFi_available* and *Parking_available* columns would be added to the output table. As a result of this process, columns to hold the values of each potential business feature are created.

Once the table structure is created, it is populated with data. First, each distinct tuple for the static properties is added as a new row to the output table. In our example, the tuples *(B1, Pete's Pizza, 4.5)*, *(B2, Sushi & Go, 3.8)*, and *(B3, Wine Heaven, 4.0)* would be initially placed in the output table. This way, rows are compacted into just one row per entity.

However, these rows are incomplete, as the newly created columns, i.e., *WiFi_available* and *Parking_available*, are not filled yet. To provide values for

these columns, the pivot operator checks, for each row and each group of columns corresponding to a pivoting value, whether the input table contains a row with the static values of that row plus the corresponding pivoting value. For instance, continuing with our example, to calculate the value of the *WiFi* column for the *(B1, Pete's Pizza, 4.5)* row, the pivot operator checks if the input table has a row containing the values *(B1, Pete's Pizza, 4.5, WiFi)*. If such a row is found, the pivot operator copies the value of their pivoted columns below the columns corresponding to the value of the pivoting properties being checked. In our case, the value of the *available* column for the *(B1, Pete's Pizza, 4.5, WiFi)* row is copied into the cell corresponding to the *WiFi_available* column. If such a row were not found, the corresponding cells would be filled with a null or blank. For instance, in our example, there is no row in the input table including the values *(B3, Wine Heaven, 4.0, Parking)*, so the *Parking_available* cell for the *(Wine Heaven, 4.0)* row is left blank.

So, summarising, when two classes are related by a multi-bounded reference, we can get the desired tabular representation by pivoting the result of the left outer join between both classes. Therefore, to ease carrying out this task, we added a primitive to Lavoisier that abstracts the described chain of operations for processing multi-bounded references. This primitive is also called *include*, for the sake of consistency. However, there is an element on this chain of low-level operations that this primitive cannot calculate by itself and, therefore, needs to be provided by the language users. This element is the set of properties of the referenced class that will be used to identify the pivoted columns. So, the *include* keyword, when applied to multi-bounded references and oppositely to single-bounded ones, must be mandatorily followed by the *by* keyword and a set of properties of the referenced class that can be used to identify their instances. Listing 6 provides an example of Lavoisier specification for including the multi-bounded reference used as example throughout this subsection.

Listing 6: Businesses data with information about their location and offered features.

```
1 dataset yelp_businesses {
2   mainclass Business
3   include location
4   include features by name
5 }
```

Finally, it is worth to mention that Lavoisier allows selecting several references at the same time, by simply using individual *include* primitives for each reference, like shown in Listing 6 (lines 3-4). In those cases, each reference would be processed independently by using the appropriate transformation (e.g. single or multi-bounded).

3.5. Aggregated Values

It could also be the case that we are not interested in analysing individually each instance of a multi-bounded reference, but in summarising the information contained in these instances by means of some metrics. For example, combining data of each review along with business data would not have too much sense from a data analysis point of view. This is, when creating a dataset with *Business* as main class, if we use an inclusion clause like *include reviews by r_id*, we would get a very big and sparse table as output. In this table, a new group of columns would be created for each individual review, and these columns would only be filled at one row, being null for the other businesses, since each review is about just one business.

Therefore, in this case, it would be more appropriate to summarise the main aspects of these reviews by means of some indicators. For example, we might be interested in knowing the values for metrics such as the number of reviews a business has received, or the number of these reviews that are positive, considering as positive those ones that at least granted a 4-stars rating.

For this purpose, Lavoisier provides the *calculate* primitive and some prebuilt aggregation functions. Listing 7 shows an example where this primitive is used for computing the two previously described metrics. In this example,

Listing 7: Lavoisier's *calculate* construct to perform aggregations.

```
1 dataset yelp_businesses {
2   mainclass Business[name, stars]
3   calculate numReviews
4     as count(reviews)
5   calculate numPositiveReviews
6     as count(reviews) where stars >= 4
7 }
```

together with the business name and stars, two columns, named *numReviews* and *numPositiveReviews*, are added to the output dataset. The first column uses the predefined *count* function to compute the number of reviews received by each business, whereas the second ones combines this predefined function with a filter to figure out the number of positive reviews for each business. Apart from *count*, the current version of the language includes the *sum*, *avg*, *max* and *min* aggregation functions, and we plan to include more of these functions as part of our future work.

3.6. Nested References

As previously commented, when including a reference, the default behaviour involves appending all attributes of the referenced class in the output dataset, whereas references are ignored. Nevertheless, we might be interested in customising which information of a reference is included to, for instance, omit certain target class' attributes, or to include some nested reference to another class.

To perform attribute selection in a reference, we can specify the concrete set of attributes to extract between brackets, just as it is done with the main class (see Section 3.2). For the purposes of including nested references, we can add a block to the *include* construct. Inside this block, we can use the same modifiers as in the main class to include references of references. These *include* blocks can be consecutively used to select references up to the required nesting level

Listing 8: Reviews data with some attribute and reference selections.

```
1 dataset yelp_reviews {
2   mainclass Review [r_id, stars]
3   include user
4   include business {
5     include location[address, postalCode]
6     include categories by name
7   }
8 }
```

for the extraction. Combining attribute selection and reference nesting, we can specify datasets with only the precise information coming from those references that we consider relevant for each concrete analysis.

In Listing 8, inside the *business* inclusion block, the *location* and *categories* references, belonging to the *Business* class, are included (lines 5 and 6). From the *location* reference, just the *address* and *postal code* attributes are selected. If the *Location* or *Category* classes had references of their own, we could also select them by using new inclusion blocks.

3.7. Inheritance Management

When a class C, like *Feature* in Figure 3, that appears in a Lavoisier specification is included in an inheritance hierarchy, Lavoisier presents the following default behaviour.

First of all, it is considered that all attributes and references of the superclasses of C are also attributes and references of such a class C, as it is usual in the object-oriented world. So, Lavoisier manages these just as regular attributes and references of the class C: simple attributes coming from superclasses are added by default, whereas references are left out if they are not explicitly added using an *include* statement.

Secondly, the attributes of the subclasses might also be of interest for some analysis scenarios. Similarly, we might wish to know to which specific subclass of

Listing 9: Selection of only two subclasses from the inheritance for the reduction.

```
1 dataset yelp_businesses {
2   mainclass Business[name, stars]
3   include features by name {
4     only as GroupedFeature {
5       include group
6     }
7   }
8 }
```

C each instance of this class belongs to. Because of these two reasons, Lavoisier also adds by default all attributes belonging to subclasses of a class C to the output dataset, as well as a new column to specify the concrete type of each instance of C. This strategy is similar to the *Single Table Pattern* used by Object-Relational Mappers (ORMs) [28].

In Lavoisier, this default behaviour can be modified, using the *as* and *only_as* primitives, with three different purposes: (1) include information about just some subclasses in the resulting dataset, excluding the other ones; (2) include some references of subclasses; or (3) select just a subset of attributes from certain subclasses, excluding the other ones.

Listing 9 shows a dataset specification example where the *only_as* construct is used. In this example, *features* (Line 3) is a reference from *Business* to the *Feature* class, which belongs to an inheritance hierarchy. To specify that we only want to include information coming from the *GroupedFeature* subclass, we indicate it using the *only_as* keyword inside an inclusion block (Line 4). Once a subclass is added to the inclusion block, we can customise it as in the case of the main class or a reference. For instance, in Listing 9, the *include* clause is used to specify that the *group* reference of the *GroupedFeature* class must be also added to the output dataset (Line 5).

If we were interested in customising a subclass without excluding other sub-

classes from the output dataset, we should use the *as* keyword, instead of *only_as*. For example, in Listing 9, if we had used *as* instead of *only_as* in line 4, both the *ValuedFeature* and *AvailableFeature* subclasses instances would be also included in the output dataset. So, using the *as* keyword, we can include references of a subclass, or exclude some of their attributes, without affecting the other subclasses.

4. Implementation

Here we include the relevant aspects with respect to the implementation of Lavoisier. Precisely, we describe the main language components, and the steps through which a Lavoisier script is processed to generate output datasets. This implementation has been open-sourced in an external repository[3].

We defined and implemented Lavoisier from scratch, instead of opting for extending any existing language. The reasoning behind this decision is the fact that we wanted to develop a language with primitives abstracted from the operations that are typically used when processing data. In our humble opinion, the extension of an existing data processing language, such as SQL, did not offer any clear advantage over the freedom provided by defining an in-house one. Moreover, typical data processing languages, such as SQL, are not able to handle high-level conceptual data representations, such as object-oriented models, which were the formalism we selected to represent data available in a domain. This issue guided us more into preferring to develop a new language, instead of extending or modifying an existing one.

4.1. Language Components

Figure 8 shows an overview of the main components that conform the language infrastructure. Lavoisier has been designed following a metamodelling approach [29, 30]. According to this approach, the abstract syntax of the DSL must be firstly defined by means of a metamodel (Figure 8, label 1). Figure 9

[3] https://github.com/alfonsodelavega/lavoisier

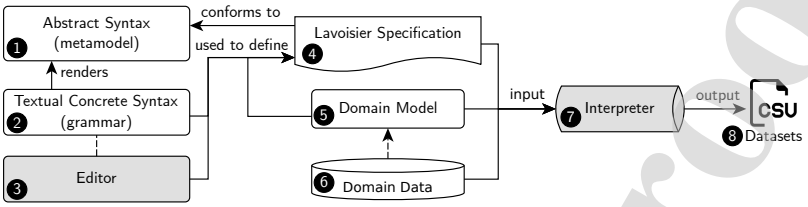

Figure 8: Main components of the Lavoisier implementation.

shows the Lavoisier metamodel, which was defined using Ecore [31]. Based on this metamodel, a Lavoisier script can include several *Dataset* specifications, each of which is defined around a *MainClass* element. From a main class, we can customise the resulting output by filtering instances with a boolean expression; selecting attributes through an *AttributeFilter*; and/or adding a set of *IncludedReferences*. Depending on the selected reference and the desired processing, there are two types of *IncludedReferences*: *SimpleReference* for single- and multi-bounded references (described in Sections 3.3 and 3.4, respectively), and *AggregatedReference* for those references that are summarised with an aggregation function (see Section 3.5). Finally, any inheritance hierarchy present in the domain model can be treated by using a *TypeFilter*, either to complete a type (*TypeCompletion*), or to limit the extraction to a subset of the existing types in the inheritance (*TypeSelection*). Section 3.7 includes some examples of how these two type filters can be applied.

After defining the abstract syntax of our DSL, we need to provide a concrete syntax, which can be either a graphical or a textual one, or even a combination of both (Figure 8, label 2). In the case of Lavoisier, we opted for creating a textual syntax because we believe that, in this dataset extraction context, writing plain text is faster than drag and drop boxes and arrows. Moreover, we also believe that, again for this context, textual specifications can be more easily understood, and might feel more familiar for end users (i.e. data scientists) than graphical ones. Nevertheless, these beliefs must be empirically checked, which will be part of our future work.

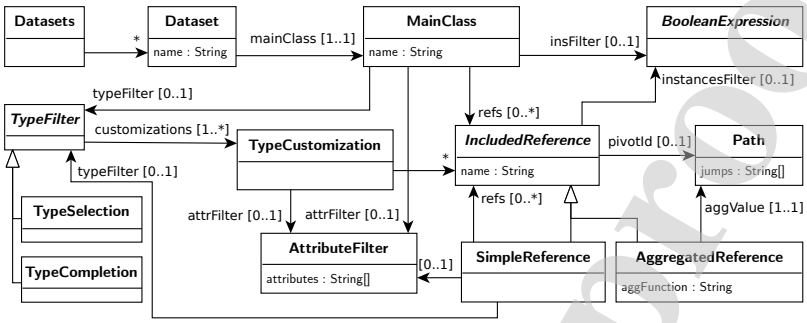

Figure 9: Abstract syntax of the Lavoisier language.

The concrete textual syntax of Lavoisier, detailed in Section 3, has been specified with a grammar in the Xtext framework [32]. Xtext is able to automatically generate, from a grammar specification linked to a metamodel, a full-fledged editor plus a parser for that grammar (Figure 8, label 3). The generated editor offers a smooth inclusion of facilities for the users, such as syntax highlighting, auto-completion, or validation during the composition of dataset specifications. For the interested reader, the current concrete syntax of Lavoisier can be consulted in the external repository of the language[4].

Using the Lavoisier editor generated by Xtext, data scientists can write data selection scripts, that would be processed by the grammar parser to represent these scripts as models conforming to the abstract syntax metamodel (Figure 8, label 4). These models are used as input for the language interpreter, which receives other two additional inputs: a domain model and the domain data. The *domain model* (Figure 8, label 5) is an object-oriented model describing data available in a domain. This domain model is also defined using Ecore. *Domain data* (Figure 8, label 6) are the actual data existing in a domain, which must conform to the *domain model*. The domain data is composed of instances of Ecore (meta)classes belonging to the domain model.

[4] https://github.com/alfonsodelavega/lavoisier/blob/master/es.unican.lavoisier/src/es/unican/lavoisier/Lavoisier.xtext

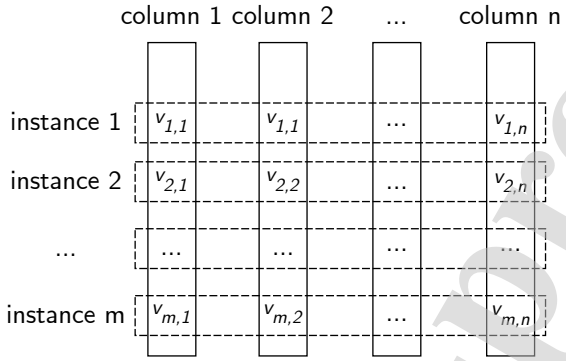

Figure 10: Structure of a *ColumnSet* object.

With these three inputs, the Lavoisier interpreter (Figure 8, label 7) generates output datasets based on the specifications contained in the Lavoisier script. The interpreter uses the domain model to know which attributes each domain entity has, and how these entities relate. This is required mainly to create the structure of the datasets. Moreover, the interpreter uses the domain data for populating the resulting datasets and, in those cases where a pivot operation is required, to determine part of the structure of the output dataset.

The *output dataset* (Figure 8, label 8) is generated conforming to the *CSV (Comma-Separated Values)* format, which is a de facto standard to provide inputs for data mining algorithms.

4.2. Execution Strategy

The execution of a Lavoisier script generates a CSV file for each dataset specification. The steps followed by the language interpreter to process a dataset specification are described in the following.

First, the instances of the selected main class that are to be included in the output dataset are gathered. Depending on the specification being processed, different tasks might take place in this step, such as filtering the instances based on a provided predicate (see Section 3.2); or taking into account an inheritance hierarchy (see Section 3.7).

```
01 dataset yelp_reviews {
02   mainClass Review
03     where business.location.city = "York"
04   include user
05   include business {
06     include location[address, postalCode]
07     include features by name {
08       as GroupedFeature {
09         include group
10       }
11     }
12     calculate numPosReviews
13       as count(business.reviews)
14       where stars >= 4
15   }
16 }
```

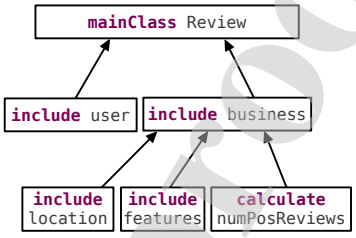

Figure 11: Left: dataset specification where information about York businesses' reviews is gathered; right: hierarchy of *ColumnSets* obtained from the dataset specification.

Once we have the selected set of instances, the output columns are calculated, and the dataset is populated. There are different sources of columns in a dataset specification. For instance, the attributes of the main class, or an included reference, are two sources of columns. Each one of these sources, when processed, generates what we denoted as a *ColumnSet* object, whose structure is shown in Figure 10. A ColumnSet is simply a two-dimensional matrix, where the rows are indexed by the selected instances from the main class, and the columns are the ones resulting from processing the associated source of columns. This structure, which resembles the one used inside a Pandas *DataFrame* object [13], allows for a seamless combination of the different ColumSets of a dataset specification up to the point of having a single ColumnSet, which would become the resulting dataset. As all the ColumnSets are indexed by the same main class instances, the process of combining two of these ColumnSets involves simply joining the rows by their index value. This structure also facilitates the extensibility of Lavoisier, as in most cases supporting a new primitive would only involve its inclusion in the syntax, and the processing of a new type of ColumnSet object.

As an example, Figure 11, left shows a dataset specification where different information about business reviews in the York city is extracted. Namely, apart from the basic *Review* attributes (line 02), this specification includes data about the *user* performing the review (line 04), and about the reviewed *business* (lines

05-15). In addition to the *Business* attributes, the specification also includes its location (line 06), the business' features (lines 07-11), and the number of positive reviews received by the business (lines 12-15).

When such a dataset specification is processed, the tree of ColumnSets that is obtained is depicted in Figure 11, right. Starting from the main class ColumnSet, there are two nested ones, one corresponding to the *user* reference, and another one for the *business* one. In turn, the *business* ColumnSet has three nested ones, related to the *location*, *features* and *numPosReviews* inclusions.

The ColumnSet tree is processed in a pre-order fashion, i.e., starting from the root, the parent node is the first one, and then the children are processed from left to right, respecting the same pre-ordering if these children have their own descendants. Once the ColumnSets are processed, they are combined following the same pre-order arrangement.

In the example, the *"mainClass Review"* ColumnSet would be the first, generating the columns {*r_id, date, stars, text*}, according to Yelp's conceptual model of Figure 1. The attributes of the *Review* instances would be consulted to populate this ColumnSet. Then, the *"include user"* ColumnSet would be next, which results in the columns {*user_u_id, and user_name*}. To populate this ColumnSet, the *user* reference of the *Review* class would be navigated for each instance, and the *u_id* and *name* attribute values would be obtained. The process would continue for the rest of the ColumnSets. When finished, these ColumnSets would be combined. For instance, when combining the two ColumnSets described above, a single ColumnSet with the columns {*r_id, date, stars, text, user_u_id, user_name*} would be obtained.

5. Evaluation

Lavoisier aims to increase the abstraction level at which data scientists work when creating datasets. To achieve this goal, Lavoisier provides different high-level primitives that, when processed, automatically execute a set of low-level operations that rearrange domain data into a tabular form that data mining

algorithms can process. This provides two main advantages: (1) data scientists do not have to write by hand boilerplate code containing long chains of data transformation operations; and (2) data scientists can get rid of the accidental complexity associated to these low-level operations. Therefore, data scientists can focus on the data to be selected, and forget about how these data will be eventually transformed.

In this section, we evaluate more systematically and objectively how much boilerplate code and accidental complexity can be reduced thanks to the use of Lavoisier. We describe first our evaluation procedure, detailing how boilerplate and accidental complexity reductions were measured. Then, we comment and discuss on the gathered results, pointing out evidences about how Lavoisier achieves its goals. Finally, threats to the validity of our conclusions are indicated and analysed.

5.1. Evaluation Method

To measure the actual benefits provided by Lavoisier, we compared it against current state-of-the-art technologies for data selection and preparation. As commented in Section 2.3, data scientists mainly use two different kinds of tools to carry out these tasks: data-management languages, and/or specialised libraries belonging to general-purpose languages. Therefore, we selected a representative of each category to be compared against Lavoisier. We chose the SQL language [11] as representative of data-management languages, and Python Pandas [13] as data-management library.

To perform the comparison, we created scripts, using Lavoisier, SQL and Pandas, for a comprehensive set of dataset extraction scenarios. Then, for each scenario, we firstly measured script size. This should provide a first and quick indicator on the Lavoisier effectiveness for reducing boilerplate code in these scripts. However, script size might be not a reliable indicator for assessing abstractness and effectiveness of Lavoisier. Size reduction could be due just to a more compact and, in some cases, more counter-intuitive syntax. A typical example would be the use of the conditional ternary operator (i.e. `condition?`

`codeForTrue : codeForFalse`) instead of opting for classical if statements (i.e. `if (condition) codeForTrue else codeForFalse`). The former is more compact, but it is debatable whether it can be more easily understood, or is more high-level, than the latter.

Therefore, to assess more rigorously whether the lower script size in Lavoisier is a consequence of using more abstract and powerful primitives, and less boilerplate code, we also calculated a set of complexity metrics. These metrics aim to measure the amount of boilerplate code and accidental complexity in a data selection script. In the following, we describe how these measures were taken, and how we designed the set of scenarios for the comparison.

5.1.1. Comparison Metrics

When measuring script size, and to prevent differences in the coding style of each language from affecting the results, we counted characters of the scripts, but ignoring any kind of whitespace and line breaks. Moreover, in the case of SQL and Pandas, we just focused on the generation of a table that contains the desired dataset, not computing all the code that would be required to store this table in a CSV file. To measure accidental complexity, we calculated the metrics described in Table 1 for each one of the developed scripts.

The rationale behind these metrics is as follows: *NumOps* is a way to measure how powerful the operations of each language are. If Lavoisier required less operations than other languages to perform the same task, it might be a symptom than Lavoisier operations can do more work per operation than the other alternatives. In this line, *NumDiffOps* tries to detect if, when using a language that requires a greater number of operations than Lavoisier, this number is caused by a need to perform a large quantity of different operations or, instead, we need to repeat the same small set of operations several times, which might indicate the presence of boilerplate code. *NumPar* and *AvgParOp* aim to quantify how complex it is to specify operations in each language. If, to perform the same task, the operations of a language require more parameters than the ones from other languages, it might mean that some parameters of the former

Table 1: Script complexity metrics.

Name	Description
NumOps	Number of operations
NumDiffOps	Number of distinct operations
NumPar	Number of parameters
AvgParOp	Average number of parameters per operation
NumKw	Number of keywords
NumDiffKw	Number of distinct keywords

one might be omitted and inferred by the operations themselves in the latter alternatives. Finally, *NumKw* is also a mechanism to measure how powerful are the abstractions of a language. As before, if a language requires fewer keywords than other to perform the same task, it might be an indicator of these keywords being more powerful and abstract. Complementary, *NumDiffKw* is a measure of the vocabulary or concepts size that a data scientist needs to manage to carry out a dataset extraction.

5.1.2. Design of the Set of Comparison Scenarios

The extraction scenarios used for performing the comparisons were carefully designed to cover a wide range of potential situations. These scenarios were designed over two different case studies: the Yelp running example, described in Section 2.1 and used throughout this paper; and an extra case study based on an online video game platform. Using a second case study allows specifying more examples to better cover some concrete cases that cannot be described using the Yelp case study. Moreover, the use of a second case study also helped provide some evidences of Lavoisier primitives not being constrained to a single case study or domain. In the following, we refer to this second example as the *VideoGames* case.

Figure 12 shows the conceptual object-oriented model for the VideoGames case-study. This model represents the different elements of a video game plat-

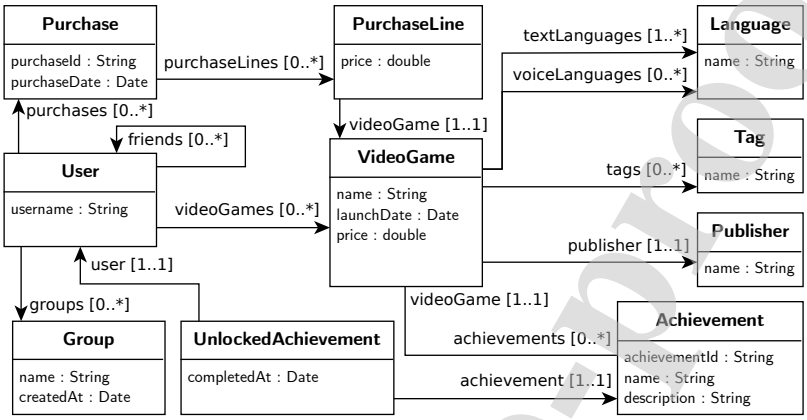

Figure 12: Conceptual model of the *VideoGames* case study.

form, and it is inspired in existing platforms such as Steam[5]. In the platform, a *User* owns a collection of *VideoGames*. Users can belong to *Groups*, and maintain a list of *friends*. In addition, users have access to their video game *Purchases*. For each *VideoGame*, data are stored about its *Publisher* company, the received *Tags* (e.g. *strategy*, *multiplayer*), which in-game textual and voice *Languages* are available, and the list of *Achievements* that users can complete while playing.

Using the Yelp and VideoGames case studies, the concrete set of scenarios described in Table 2 was finally selected to perform the comparisons. These scenarios, as already commented, focus on discovering how the different structures that can be found in an object-oriented model are managed by each language. Each scenario was labelled with a code, composed of a single character plus a number. The character indicates the kind of transformation pattern being analysed: *a* refers to the trivial single class case, described in Section 3.1; *b* are those cases where a single-bounded reference is included in the main class, while *c* cases do the same for multi-bounded references (see Sections 3.3 and 3.4,

[5] https://store.steampowered.com/

Table 2: Dataset extraction scenarios performed in the comparison.

Code	Case	Model	Description
a	Single table/class	VideoGames	Just *VideoGame*
b1	Single-bounded reference	Yelp	*Reviews* and their *user*
b2	Single-bounded reference	VideoGames	*Achievements* and their *videoGame*
c1	Multi-bounded reference	Yelp	*Business* and their *categories*
c2	Multi-bounded reference	VideoGames	*VideoGame* and their *text languages*
d1	Inheritance	Yelp	*Features* (relational single table)
d2	Inheritance	Yelp	*Features* (relational concrete table)
d3	Inheritance	Yelp	*Features* (relational class table)
d4	Multi-bounded Inheritance	Yelp	*Businesses* and their *features* (relational single table)
d5	Multi-bounded Inheritance	Yelp	*Businesses* and their *features* (relational concrete table)
d6	Multi-bounded Inheritance	Yelp	*Businesses* and their *features* (relational class table)
e1	Combination (multiple)	VideoGames	*VideoGames* with *tags* and *publisher*
e2	Combination (multilevel)	Yelp	*Reviews* with their *business* and the *categories* of such business

respectively); *d* refers to those cases involving inheritance (see Section 3.7); and *e* tests the capabilities to process several references at the same time, or to deal with nested references (see Section 3.6). Besides each scenario, a description of the classes to be included in each dataset is provided. For instance, in scenario *c1*, the main class would be *Business*, and the multi-bounded reference *categories* would be included.

Finally, it is worth to mention that Lavoisier is able to directly perform dataset extractions over conceptual models. However, SQL and Pandas need to work at the relational or table-level. Therefore, we derived the relational models associated to the conceptual models of the Yelp and VideoGame case studies. This transformation process was straightforward, except for the cases where inheritance was present. Inheritances can be transformed into a relational model using typically three strategies, known as *Single Table*, *Concrete Table*, and *Class Table* (also known as *Joined Mapping*) [28]. Therefore, for those scenarios that tackle inheritance, we created three different relational models, each one following a different inheritance mapping strategy; and we performed the comparison against each one of these alternatives.

5.2. Dataset Extraction Example

Before commenting on the obtained results, we detail here the scripts for a concrete dataset extraction scenario. The goal of this section is to provide the reader with a more clear vision of how the scripts we used for the evaluation work in Lavoisier, SQL and Pandas. For this purpose, we will use the *d5* scenario (see Table 2), where a dataset of businesses is generated including the information about their features. Figure 13 shows the domain model fragment that needs to be retrieved by Lavoisier on the left, and the relational counterpart processed by SQL and Pandas on the right.

In the *d5* case, the inheritance hierarchy of the domain model is translated to the relational model using the *Concrete Table Inheritance* mapping pattern [28]. When following this pattern, a table is generated for each non-abstract class in the inheritance tree. In this case, three tables are generated: *AvailableFeature*,

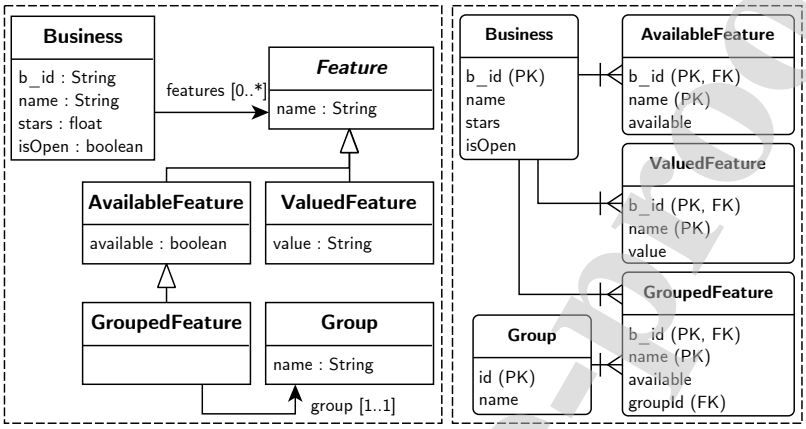

Figure 13: Left: domain model fragment queried by Lavoisier in the *d5* case; right: equivalent relational database queried by SQL and Pandas.

ValuedFeature, and *GroupedFeature*. The feature name, which belongs to the abstract class, is included in each one of these tables. Moreover, these tables have foreign key references to the business they belong to, since the multi-bounded reference between the *Business* and *Feature* classes must be preserved. This mapping is equivalent to replacing the *features* reference in the domain model with three new references, each one from the *Business* class to a different *Feature* subclass. In addition, the *GroupedFeature* table has a foreign key to the *Group* table.

Listing 10: Lavoisier script to process the domain model of Figure 13, left.

```
1 dataset businessAndFeat {
2   mainclass Business
3   include features by name {
4     as GroupedFeature {
5       include group
6     }
7   }
8 }
```

First, Listing 10 shows the Lavoisier script for this scenario. This script specifies that the attributes of the *Business* class must be included in the dataset, as well as the *features* reference (lines 2 and 3). Since this is a multi-bounded reference, instances of the *Feature* class are spread over columns using the *name* attribute to distinguish between these instances (see Section 3.4). Moreover, as part of the features inheritance, the *group* reference of the *GroupedFeature* subclass is also included (lines 4-6). As it can be noticed, a data scientist writing this script would focus on the data to be selected, and no information about how these data should be rearranged, except for the feature name attribute, needs to be provided.

Listing 11: Pandas script to process the relational database of Figure 13, right.

```
1  gfs = (groupedFeatures
2         .merge(fGroups.rename(columns={"name" : "group_name"}),
3                left_on="groupId", right_on="id")
4         .drop("groupId", axis=1)
5         .pivot(index="b_id", columns="name",
6                values=["available", "group_name"]))
7  gfs.columns = ['_'.join(col[::-1]).strip() for col in gfs.columns.values]
8
9  (businesses.merge(availableFeatures.pivot(index="b_id",
10                                           columns="name",
11                                           values="available"),
12                   left_on="id", right_on="b_id", how="left")
13   .merge(valuedFeatures.pivot(index="b_id",
14                               columns="name",
15                               values="value"),
16          left_on="id", right_on="b_id", how="left")
17   .merge(gfs, left_on="id", right_on="b_id", how="left"))
```

Second, we analyse the Pandas script for this scenario, which is shown in Listing 10. Pandas loads each relational table in a tabular data structure denoted as *dataframe*. In Listing 10, each table is already loaded in a different dataframe. In lines 1-3, the dataframes holding the *GroupedFeature* and *Group* instances are combined by means of a *merge* operation, which is the Pandas equivalent of a join in SQL. Before performing this operation, we need to rename the *name* attribute of the *Group* table, loaded in the *fGroups* dataframe, to avoid collisions with the *name* attribute of the *GroupedFeature* table. More-

over, in Pandas, when joining two tables, or dataframes, all columns of the joined tables are combined. So, any unwanted column needs to be manually *dropped* after the merge. This can be seen in line 4, where the *groupId* column of the *GroupedFeature* table is dropped after the merge of lines 2-3.

After combining each *GroupedFeature* with its *Group*, we would need to combine each business with its set of features. This is, we need to process the multi-bounded references (in this relational example, one-to-many foreign keys) existing between the *Business* table and each *Feature* table. According to Section 3.4, to process these references we need to combine a join with a pivot operation. So, we pivot each feature table using as pivoting attribute the feature name (lines 5-6, 9-12 and 13-16) to spread features belonging to the same business over columns. Then, each pivoted table is merged with the *Business* table (lines 9, 13 and 17), which is hold in the *businesses* data frame.

The first noticeable difference with Lavoisier is that each subclass of the *Feature* class needs to be processed individually, as they are placed in separated tables. On the contrary, in Lavoisier we only need to include the reference to the superclass, and all the child classes are automatically included. This can be advantageous in cases where a class has a high number of subclasses. However, it is true that if the subclasses have references that might be included in the dataset, we would need to handle these references individually with both languages.

A second difference is that, in Lavoisier, references are just included and the language takes care of performing the required data transformations transparently to the data scientist. On the other hand, in Pandas we need to transform the data manually, by means of combining joins and pivot operations. Moreover, these operations need some extra parameters that in Lavoisier are not required. For instance, merge operations require the specification of the identifier columns of each table to be merged, and the kind of merge to be applied (e.g. *inner*, *left*). The pivot operation needs the explicit specification of the *static*, *pivoting* and *pivoted* sets of columns. In Lavoisier, many of these parameters are automatically inferred. For instance, static and pivoted columns are

automatically calculated using the attributes of the class and the set of pivoting columns. So, data scientists need to deal with less parameters when using Lavoisier.

Furthermore, data scientists must pay attention to some picky details when using Pandas, such as renaming attributes to avoid name collisions. These small issues are also automatically handled by the Lavoisier interpreter, so the data scientist does not need to care about them.

Listing 12: SQL script to process the relational database of Figure 13, right.

```
1  select * from business b
2  left join
3  (select af.b_id,
4   max(case af.name when 'Parking' then available end) as feat_parking,
5   max(case af.name when 'WiFi' then available end) as feat_wifi
6  from availableFeature af
7  group by af.b_id) afs on b.id = afs.b_id
8
9  left join
10 (select vf.b_id,
11  max(case vf.name when 'Smoking' then value end) as feat_smoking,
12  max(case vf.name when 'AgesAllowed' then value end) as feat_agesAllowed
13 from valuedFeature vf
14 group by vf.b_id) vfs on b.id = vfs.b_id
15
16 left join
17 (select gf.b_id,
18  max(case gf.name when 'Breakfast' then available end) as
       feat_breakfast_available,
19  max(case gf.name when 'Breakfast' then g.name end) as
       feat_breakfast_groupName
20 from groupedFeature gf
21 inner join Group g on gf.groupId = g.id
22 group by gf.b_id, g.name) gfs on b.id = gfs.b_id;
```

Lastly, we show the SQL script for this scenario in Listing 13. As in the Pandas case, we need to process each feature table individually, and combine them with the *Business* table using joins and pivots. However, the SQL standard does not include a version of the pivot operator, although it can be found in some SQL dialects. Therefore, to simulate pivots in SQL, we decided to use the workaround already described in Section 2.3, because it can be used in any

Listing 13: SQL Server's pivot operation example.

```
1  select bName, stars, "Wifi", "Parking"
2  from (
3      select b.name as bName, b.stars,
4             f.name as fName, f.available
5      from Business b, Feature f
6      where f.business = b.id
7  ) as p
8  pivot (
9      max(p.available)
10     for p.fName in ("Wifi","Parking")
11 ) as pivotTable;
```

flavour of the SQL language. When using this workaround, we need to create the pivoted columns manually, and then, using an aggregation function plus a case statement in the context of a group by clause, calculate the value for each pivoted column. This means we need to know, at the time of writing the script, all possible values that the pivoting columns might hold.

For this scenario, we have considered that businesses might have *Parking* and *WiFi* as available features, *Smoking* and *AgesAllowed* as valued features, and *Breakfast* as grouped feature. Lines 4 and 5 show how the pivoted columns *feat_parking* and *feat_wifi* are manually created when pivoting the *Available Feature* table. Similar columns are added for the *Valued Feature* and *Grouped Features* tables in lines 11-12 and 18-19, respectively. As it can be seen, script complexity increases in the SQL case when compared to Pandas.

It is worth to mention that, even in the lucky cases where a SQL dialect provides an implementation of the pivot operator, this often comes with some limitations. For example, SQL Server extends the SQL standard with a proprietary version of the pivot operator [16], where the structure of the output table is not calculated by the operator itself, but instead it has to be manually specified. Listing 13 illustrates how a single multi-bounded reference can be

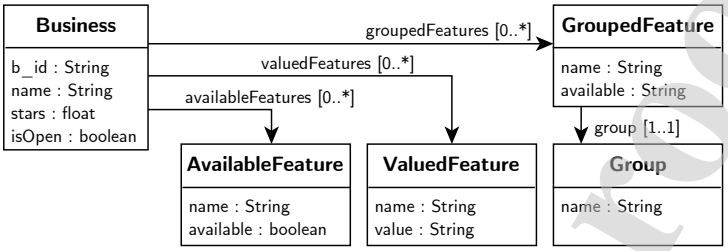

Figure 14: Alternative domain model of business and features, without using inheritance.

managed in SQL Server by pivoting the result of a join between two tables, which in this case would be the *Business* and *Available Feature* tables. As it can be seen, in the pivot operation (lines 8-11), we must explicitly specify the values of the pivoting properties for which new columns would be generated (Line 10). It must be noticed that, if new feature values are added to the system, we must update this script to deal with them, which is a less maintainable solution as compared to other implementations of the pivot operator, such as the one provided by Pandas, where these values are automatically calculated. This solution, where values of pivoting columns must be explicitly listed, might be also problematic when the number of these values is too high.

Finally, it could be argued that this dataset extraction scenario benefits Lavoisier particularly, as it can handle the inheritance hierarchy transparently, whereas Pandas and SQL need to deal with three separated tables. First, we consider the ability to work against a higher level data representation (i.e. object-oriented domain models) one of Lavoisier's main advantages. Nonetheless, to see how Lavoisier would cope with this scenario if there were not any inheritance tree, we analysed an alternative domain model illustrated in Figure 14. In this model, there is no *Feature* abstract class, and the *Business* class has a multi-bounded reference to each feature kind. Listing 14 contains the Lavoisier script required to process this new domain model.

As with SQL and Pandas, in this case Lavoisier needs to include references to each feature class individually (lines 3-7). However, this selection only takes

Listing 14: Lavoisier script to process the domain model of Figure 14

```
1 dataset businessAndFeat {
2     mainclass Business
3     include availableFeatures by name
4     include valuedFeatures by name
5     include groupedFeatures by name {
6         include group
7     }
8 }
```

one line for the *AvailableFeature* and *ValuedFeature* classes (lines 3 and 4, respectively), and three lines for the *GroupedFeature* one (lines 5-7), as the *group* reference also needs to be selected. Oppositely to Pandas and SQL, data scientists would not need to care about pivoting and joining these references, since these operations are automatically executed by the language interpreter when processing the *include* primitive. Similarly, data scientists do not need to care about name collisions. Moreover, each Lavoisier inclusion is independent, i.e., when including a reference in Lavoisier we do not need to care about the previous or subsequent inclusions (see Section 3.4, last paragraph). On the contrary, when using SQL and Pandas every new data bundle (e.g. a query over a table) needs to be joined with the previously selected data, which is tedious and might lead to errors.

5.3. Results and Discussion

Figure 15 shows the character-size of the scripts created with Lavoisier, SQL and Pandas for each evaluation scenario.[6]. In addition, Table 3 contains complexity metrics values for each script, and the average percentage reduction obtained for each metric when using Lavoisier instead of the other technologies.

[6]These scripts are available in an external repository: `https://github.com/alfonsodelavega/lavoisier-evaluation`.

Table 3: Complexity metrics values per scenario and extraction technology, including in the last row the average reduction for each metric obtained when using Lavoisier (Lv) with respect to SQL and Pandas.

	NumOps			NumDiffOps			NumPar			AvgParOp			NumKw			NumDiffKw		
Case	SQL	Pandas	Lv	SQL	Pandas	Lv	SQL	Pandas	Lv	SQL	Pandas	Lv	SQL	Pandas	Lv	SQL	Pandas	Lv
a	1	1	1	1	1	1	1	1	1	1,00	1,00	1,00	2	0	2	2	0	2
b1	2	2	2	2	2	2	6	5	2	3,00	2,50	1,00	6	4	3	6	4	3
b2	2	2	2	2	2	2	6	5	2	3,00	2,50	1,00	6	4	3	6	4	3
c1	6	4	2	4	4	2	9	9	3	1,50	2,25	1,50	28	8	4	14	8	4
c2	6	4	2	4	4	2	9	8	3	1,50	2,00	1,50	28	7	4	14	7	4
d1	3	3	3	2	2	3	9	9	3	3,00	3,00	1,00	9	8	6	6	5	5
d2	6	5	3	3	4	3	12	9	3	2,00	1,80	1,00	18	9	6	8	7	5
d3	9	6	3	3	4	3	21	12	3	2,33	2,00	1,00	27	12	6	8	7	5
d4	12	13	4	4	5	3	18	35	3	1,50	2,69	0,75	51	36	6	14	7	5
d5	17	11	4	4	5	3	25	28	3	1,47	2,55	0,75	62	33	6	14	14	5
d6	20	14	4	4	5	3	34	37	3	1,70	2,64	0,75	71	42	6	14	14	5
e1	8	5	3	5	4	2	23	12	4	2,88	2,40	1,33	32	11	5	15	9	4
e2	9	5	3	5	4	2	16	13	4	1,78	2,60	1,33	34	12	5	15	8	4
%Red.	46,3	36,5		18,8	24,6		70,3	66,6		41,6	49,8		68,4	47,3		51,8	35,6	

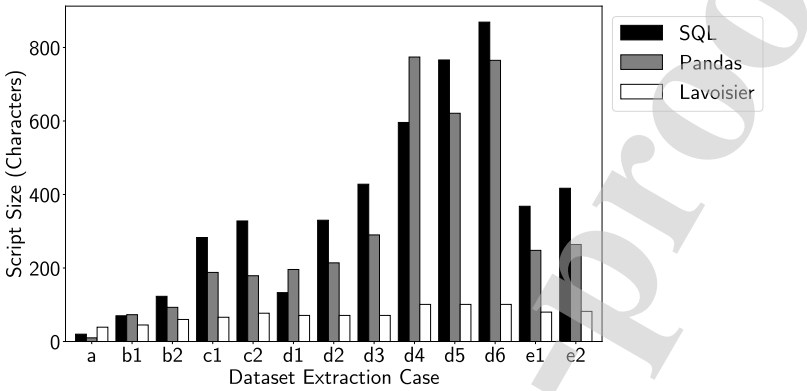

Figure 15: Script size in characters of the extractions for each approach (a: single table case; b: unary reference; c: multi-bounded reference; d: inheritance; e: combination).

On a first glimpse to Figure 15, it can be seen that the script size for both SQL and Pandas considerably increases as the scenario complexity grows. This phenomenon can be clearly appreciated in scenarios *d3-d6*, *e1* and *e2*. In contrast, Lavoisier's script size grow is steadier across all cases. On average, using Lavoisier grants a 60% script size reduction with respect to SQL, and a 40% reduction when compared with Pandas.

Analysing each case more in detail, it can be noticed that in the trivial *a*-coded cases, there is practically no difference between languages, being Lavoisier a little more verbose. In SQL, this trivial case can be addressed using a simple *SELECT* statement, whereas in Pandas is even simpler, as we only need to write the name of the table, or data frame, whose data we want to get. Oppositely, Lavoisier specifications contain additional information about the name of CSV file to be generated, whereas this information was not computed in SQL and Pandas.

For *b*-coded cases, SQL and Pandas scripts are slightly larger than those of Lavoisier. These cases refer to the inclusion of single-bounded references, which is achieved by using an *include* clause in Lavoisier, a *join* operation in SQL,

and a *merge* operation in Pandas. The *join* and *merge* operations are a little more complex than the *include* primitive of Lavoisier, as they require knowing what columns of the tables to be joined will be used for matching rows. On the other hand, in Lavoisier, we only need to specify the reference that should be included in the output dataset. In addition, as we saw in the example of the previous section, sometimes SQL and Pandas users have to manually address the following issue: column name management. When joining two or more tables, if a pair of these have the same name in some of their columns, users have to manually specify an alias to differentiate them in the resulting joined table. As another example, in the *b2* scenario, the *Achievement* and *VideoGame* entities of the *VideoGames* case study are joined, but both entities have a *name* attribute. So, this pair of attributes must be aliased, either with the *as* keyword in SQL or with the *suffixes* merge parameter in Pandas. Oppositely, Lavoisier's *include* construct makes implicit this renaming, preventing data scientists from having to worry about these low-level details.

Regarding complexity metrics of the same *b2* scenario, it should be noticed that the number of operations is the same for the three cases, since we only need to perform an additional operation in each: a reference inclusion in Lavoisier, a *join* in SQL, and a *merge* in Pandas. However, the need to specify the columns for matching rows in SQL and Pandas, plus the manual management of name collisions, lead to an increase in the values of the *NumPars* and *AvgParOp* metrics for these technologies. Moreover, the metrics related to number of keywords suffer a greater increase in SQL and Pandas. In Pandas, following the convention of the Python language, the parameters of the invoked functions are named to allow specifying only the required ones, and in the order that is preferable for the user. So, we need to add extra keywords when certain parameters are required. In the same line, SQL employs a generous amount of keywords to declare the operations and to, for instance, rename columns. This verbosity in SQL is not necessarily bad, as SQL specifications might become more readable for the non-experienced reader.

For multi-bounded references, this is, *c* cases, the script size for SQL and

Pandas is noticeably worse, as shown in Figure 15. To process this kind of references with these technologies, we need to execute a join operation, and then a pivot (see Section 3.4). So, multiple operations need to be combined along with their parameters to achieve the required transformation. In the case of Lavoisier, just the name of the reference and the set of attributes to be used as pivoting properties are required. The previous example of Section 5.2 gives more details about joining and pivoting multi-bounded relationships in SQL and Pandas.

Regarding complexity metrics in *c* cases, the number of operations is higher in SQL and Pandas, due to the need of concatenating joins and pivots. The number of different operations is also greater than in Lavoisier, which means that a single Lavoisier operation does the work of several low-level operations in SQL and Pandas. The number of parameters also increases in SQL and Pandas, aggravated by the need of performing more operations, which usually require more parameters than Lavoisier's. In the SQL case, both the total and distinct number of keywords grow noticeably too, as an effect of SQL not having a proper implementation of the pivot operator (see Listing 1).

In the case of inheritance scenarios (*d1*-*d6*), the performance of SQL and Pandas is also clearly worse. In general, SQL and Pandas need to perform several operations to compact the inheritance hierarchy, which adds a lot of boilerplate code in these cases. For instance, in SQL, several joins might be required to combine subclasses with superclasses. These operations are automatically carried out by Lavoisier, so data scientists do not need to deal with them. This obviously leads to an increase in script size, which is specially noticeable in the *d3* to *d6* scenarios. These scenarios correspond to the case where we want to process a multi-bounded reference pointing to a class that is part of an inheritance tree. In these cases, the inheritance hierarchy needs to be compacted several times, once per leaf in the inheritance hierarchy, which contributes to increase the amount of boilerplate code associated to these tasks in SQL and Pandas. In addition, it can be observed that the script size in SQL and Pandas seems to be independent of the strategy used for mapping the inheritance.

Only SQL might slightly benefit from the use of the *Single Table* mapping, as there are fewer tables to combine. Section 5.2 depicted the *d5* scenario, where a *Concrete Table* mapping is applied.

Considering complexity metrics in *d* scenarios, it can be observed that the number of operations grows noticeably, but the number of different operations remains stable as compared to previous scenario types. This seems to indicate the presence of boilerplate code in SQL and Pandas, as the same operation needs to be applied several times to perform a single task, such as traversing an inheritance hierarchy to find an attribute. As a consequence of the increase in the number of operations, the number of parameters and keywords also augments. On the other hand, the number of different keywords is similar to the scenarios for processing single and multi-bounded references. This is logical, as inheritance hierarchies are represented in relational databases by means of connecting tables through foreign keys. Therefore, they are processed with the same operations used in the *b* and *c* cases.

Finally, we comment on the *e1* and *e2* scenarios, which can be considered as combinations of simpler cases. In *e1*, several references are included at the same time, which would be similar to processing two or more *b* and *c* scenarios. The *e2* case involves nested references, which can be viewed as the problem of performing two consecutive reference inclusions.

For these scenarios, one detected benefit of Lavoisier is that reference inclusions in the dataset specifications are independent one of another. For instance, in the *e1* case, data about *VideoGames* are extracted, along with their *publisher* and *tags* references. In Lavoisier, each reference is selected through the use of an *include* construct, and neither of these two constructs needs to be aware of the other one. Oppositely, in SQL and Pandas, one of the references would be processed first, producing an intermediate table. Then, to handle the second reference, we must work over this intermediate table. Therefore, we have to pay attention to the order in which references are processed. Moreover, if one reference is removed from the output dataset, this removal might affect the transformations used to process the remaining ones. An extra example of

the better support of Lavoisier when processing a combination of references is shown in Listing 14, which processes the domain model of Figure 14. In that model, the features inheritance has been replaced by individual types for each concrete feature class.

In Figure 15, for *e1* and *e2* scenarios, script size increases clearly in the SQL and Pandas cases, as more intermediate operations with its corresponding boilerplate code are required; whereas script size remains stable in the Lavoisier case. Regarding complexity metrics, as before, the number of total and different operations increases, meaning than for each operation in Lavoisier, several different low-level operations need to be applied repeatedly. As the number of operations grows, the number of parameters and keywords also grows.

In summary, it can be concluded that Lavoisier contributes to increasing the level of abstraction at which data scientists work when creating datasets, by providing a set of high-level and powerful primitives. Each one of these primitives do the work of several low-level data transformation operations of alternative technologies. As a consequence, data scientists using Lavoisier need to know how to use 18% and 24% fewer different operations to create a dataset when compared with SQL and Pandas, respectively. Moreover, Lavoisier allows data scientists to get rid of boilerplate code, such as having to avoid name collisions or to traverse inheritance hierarchies. This, together with the higher abstraction level of the operations, lead to a 46% and 36% reduction on the total number of operations needed in Lavoisier. In addition, Lavoisier operations need less information to work than the low-level operations used in SQL and Pandas. More specifically, Lavoisier specifications contain around 65-70% fewer parameters than their corresponding SQL and Pandas versions. These reductions translate into the previously mentioned 60% and 40% smaller script size on average when using Lavoisier with respect to SQL and Pandas' counterparts. This size reduction can be better noticed as the number of entities to be included in a dataset grows; or when these entities are involved in inheritance hierarchies. Therefore, it can be stated that Lavoisier helps data scientists to write more abstract and powerful code and to avoid boilerplate code.

Moreover, these reductions in number of operations and parameters are also reflected in a reduction of the number of keywords required to create a dataset by 68% and 47% over SQL and Pandas, respectively. Therefore, it can be stated that data scientists need to care about fewer details, and deal with a lower accidental complexity when working with Lavoisier. The lower accidental complexity might also lead to decrease the learning curve for Lavoisier, as compared with SQL or Pandas, but this hypothesis should be confirmed empirically by means of some controlled experiments to be undoubtedly stated.

5.4. Threats to Validity

In the first place, it might be argued that results are due to the selection of the dataset extraction scenarios, and that other selection might have lead to different results. These scenarios were not arbitrarily selected, but with the objective of covering all potential scenarios that we might face when creating a dataset from hierarchical and nested data represented by a domain model. Moreover, these scenarios were kept simple, this is, we have not created artificially complex scenarios that, according to the gathered results, would benefit Lavoisier. For instance, we have not included any scenario containing large chains of references, or very large inheritance hierarchies, which are cases where Lavoisier would have played clearly better.

Secondly, it could be considered that results are biased due to the selected case studies, and that other case studies would have returned a different outcome. The *Yelp* and *Videogame* case studies were selected just for giving some semantics to the data structure to be reduced, and for making them easier to understand. Other case studies would have lead to the same results, since Lavoisier just processes the syntactic structure of the data to be transformed, so what a class or an attribute represents does not matter.

Thirdly, it can be considered that the comparison between Lavoisier, SQL and Pandas is not fair because Lavoisier works against an object-oriented domain model, whereas SQL and Pandas do it against a relational model. Therefore, Lavoisier benefits of using a more high-level input model. The latter is

true, and this is the reason why we decided to use an object-oriented conceptual model as input for Lavoisier.

Finally, the different metrics calculated and analysed in previous section provide some sound and objective evidences about the benefits of using Lavoisier for performing data selection and preparation tasks. Nevertheless, these evidences have not been rigorously confirmed by means of controlled empirical experiments. Previous studies have shown that a well-performed evaluation that includes controlled experiments with end users is one of the best ways to demonstrate the benefits of a DSL over a GPL [33, 34]. For instance, the obtained results suggest that the learning curve of Lavoisier for a new user would be better than the one for SQL or Pandas, as there are less operations to learn and these are simpler to use (e.g. they require less parameters), but we would need to check this hypothesis empirically.

At the time of writing this paper, we have just performed controlled empirical experiments for a framework called FLANDM [35, 27], which allows creating DSLs for other stages of a data mining process, and that is complementary to Lavoisier, as follows. DSLs created with FLANDM automate the execution of data mining algorithms, taking care of the fine-grained configuration of these algorithms. As input for these algorithms, FLANDM might use the datasets generated by Lavoisier. The results of the controlled experiments using FLANDM were really good, which seems to indicate that building DSLs for automating data mining tasks can provide actual benefits.

Nevertheless, as we could confirm during the design and execution of the experiments for FLANDM, several difficulties arise when wanting to perform these empirical evaluations, such as the availability of adequate participants, or the resource-intensive preparation of the experiments. These difficulties are noticeable in the DSL research community, given the clear lack of existing DSL evaluation research performing empirical experiments with end users [36]. According to our previous experience [27], these empirical experiments can require half a year of full time working, so this is the reason why they were left out as part of our future work. We will try to overcome these difficulties and perform

such empirical experiments with Lavoisier in order to empirically confirm the benefits that can expected from the objective metrics we have gathered and analysed.

6. Related Work

To the best of our knowledge, this is the first language designed to select data from domain models and generate tabular datasets from them. Currently, dataset extraction processes are usually performed by using SQL-like languages [11]; frameworks for data management that typically include their own languages, such as the R project [12] or Julia [37]; or libraries developed for general purpose programming languages, e.g., the Pandas library for Python [13] or Weka for Java [8]. As we have seen in the comparison of the previous section, using these state-of-the-art tools involves manually combining low-level operations to produce the required tabular structures, which might become a cumbersome and prone-to-errors process.

Apart from manual approaches, there is a research field aiming to automatically reduce multi-relational data to a single-table structure that can be digested by data mining algorithms, which is known in the community as *propositionalisation* [38, 39, 40, 41]. Generally speaking, these approaches work as follows: starting from an entity of interest, e.g., Business, an algorithm randomly generates dataset columns by applying aggregation functions, such as *count*, *average*, *max* or *min*, over the relationships between the selected entity and other ones in the model.

This random exploration has the potential to discover previously unknown aggregate values that might be relevant for data analysis. On the other hand, domain experts and data scientists cannot have a fine-grained control of which data would be included in the output dataset. In addition, multi-bounded references cannot be analysed at the instance level, since their information needs always to be summarised by means of aggregation functions. This limitation might hamper finding patterns related to values of these individual instances.

Moreover, it should be taken into account that this random exploration might exhibit scalability and performance problems. The exploration takes place over an enormous search space of candidate columns, from which many of them may not be useful at all.

Other researchers have tackled the problem from a different angle. Instead of focusing on producing tabular datasets from linked and hierarchical data, they have modified some data mining algorithms so that they accept data in their original structure as input, which is known as *Multi-Relational Data Mining (MRDM)* [42]. Solutions based on MRDM have been proven useful to perform analysis over relational datasets coming from diverse domains, such as medicine [43], financial [44] or time-sequence analysis [45, 46]. However, at the time of writing this work, most of these modified algorithms are not as powerful, versatile and efficient as those data mining algorithms that work with single-table datasets. While this is the case, we consider that a language like Lavoisier can be helpful for assisting data scientists in the generation of tabular datasets.

Lastly, different query languages already exist that are able to query hierarchical structures such as the one defined in a domain model. Examples of these are, among others, GQL[7], Gremlin [47], GraphQL[8], Cypher[9], and XPath[10]. Some similarities can be detected between the syntax of these languages and Lavoisier. For instance, both Lavoisier and GraphQL make use of the same brace structure in their queries to represent nesting of the queried elements (see Section 3.6). On the flip side, a commonality of all these existing languages is that, while they are able to query graph-like structures [48], the result of a query is either a value/array of values, or a structure that still follows the same graph-like structure, instead of the tabular format desired in the data selection context of this paper. Even when some export-to-table functionality is offered[11],

[7] https://www.gqlstandards.org/
[8] https://graphql.org/
[9] https://neo4j.com/developer/cypher-query-language/
[10] https://www.w3.org/TR/xpath/all/
[11] https://neo4j.com/docs/labs/apoc/current/export/csv/

the resulting table does not respect the *one entity, one row* constraint imposed by data mining algorithms discussed in Section 2.3. So, extra post-processing of the retrieved table would be required. Therefore, Lavoisier provides an advantage over these graph query languages in the context of a data-mining process, which is being able to automatically generate datasets that can be digested by data mining algorithms, without requiring any extra post-processing.

7. Summary and Future Work

This work has presented Lavoisier, a language for assisting data scientists during the creation of datasets according to the format accepted by data mining algorithms. We started by presenting the data selection and transformation problem, which states that data mining algorithms can only receive data arranged in a specific tabular format. Therefore, before executing a data mining algorithm, we need to select and rearrange any hierarchical and linked domain data of interest for the analysis into the accepted format. Data scientists typically perform this task by writing scripts in a data management technology, such as SQL or Pandas, which involves performing several low-level data transformation operations manually and taking care about their details. This leads to the creation of large and complex scripts, which is a time-consuming and prone-to-errors task.

As a solution to alleviate this problem, we have created a language that provides a set of high-level constructs for selecting data from object-oriented domain models. These constructs, when processed by the language interpreter, are transformed into a set of low-level data transformation operations that generate tabular datasets ready to be digested by data mining algorithms. Lavoisier is able to deal with the different transformation scenarios that can be found in an object-oriented model, such as the processing of single- or multi-bounded references, nested structures, or inheritance hierarchies.

When compared against typical data management technologies used for dataset creation, Lavoisier allows: (1) reducing the total number of operations

required to generate a dataset by ∼40%; and (2) decreasing also the total number of parameters to be specified in these operations by ∼65%. As a consequence of these and other reductions, the size of Lavoisier's scripts decrements by 60% and 40% with respect to the counterparts of the compared technologies (i.e. SQL and Pandas), and of up to 80% in some cases. Moreover, by using Lavoisier instead of these two technologies, data scientists need to use, on average, 18% and 24% fewer types of operations, 68% and 47% less keywords, and ∼40% less parameters per operation. This means that data scientists need to know fewer primitives to write a dataset extraction script; and they need to be aware of fewer details when using each primitive, this is, they can work at a higher level of abstraction, and with less accidental complexity.

As future work, as already mentioned in the paper (Section 5.4), we would like to perform some empirical experiments to assess Lavoisier' usability, learning curve and effectiveness. In this paper, we have provided objective and sound evidences about the effectiveness of Lavoisier for reducing boilerplate code and accidental complexity in data selection scripts (see Section 5.3). Nevertheless, we would like to go one step further and perform controlled empirical experiments, where a heterogeneous group of data scientists complete some data extraction tasks using both Lavoisier and common data management languages, such as Pandas, R or SQL. Although Lavoisier has been welcomed by the data scientists we have shown it, these empirical experiments would allow us: (1) to get better insights about how different kind of data scientists interact with Lavoisier; (2) to quantify more precisely the benefits of Lavoisier; and, (3) to identify any flaws to be addressed as part of our future work.

Secondly, we would like to explore how Lavoisier might be extended to work with other approaches for building conceptual data models. In this paper, we have used object-oriented models as approach for specifying data available in a domain, since this is a widely used and accepted approach for this task. However, other people might prefer other alternatives, such as ontologies [49], entity-relationship models [50], and different kinds of multidimensional models commonly used for data warehouse design [51].

Lastly, we would like to slightly improve the current implementation of Lavoisier. At the time of writing this paper, Lavoisier is provided as an academic prototype that supports the features described in this paper. Nevertheless, as commented in Section 3.5, the current support for aggregation functions and operators is limited. Therefore, we would like to incorporate in the near future more of these functions to Lavoisier, as well as a richer set of operators for working with dates and strings. With some of these improvements, Lavoisier would be more prepared for its use in real settings. We have plans for using Lavoisier inside the Educational and Industry 4.0 domains. We have been working in the educational data mining field for some time, so we want to check more in-depth how Lavoisier can help us in real projects. Regarding Industry 4.0, there is a non negligible amount of industrial engineers willing to apply data mining techniques for the analysis of data coming from manufacturing processes. Therefore, we believe that Lavoisier can help simplify the use of data mining techniques in this specific context.

Acknowledgements. This work has been funded by the Spanish Government under grant TIN2017-86520-C3-3-R. Some icons in Figures 2 and 8 were created by Smartline from Flaticon.

8. References

[1] C. Peng, M. Deng, L. Di, W. Han, Delivery of agricultural drought information via web services, Earth Science Informatics 8 (3) (2015) 527–538. doi:10.1007/s12145-014-0198-7.

[2] R. S. Santos, S. M. F. Malheiros, S. Cavalheiro, J. M. P. de Oliveira, A data mining system for providing analytical information on brain tumors to public health decision makers, Computer Methods and Programs in Biomedicine 109 (2013) 269–282. doi:10.1016/j.cmpb.2012.10.010.

[3] M. R. Kamdar, D. Zeginis, A. Hasnain, S. Decker, H. F. Deus, ReVeaLD: A user-driven domain-specific interactive search platform for biomedical

research, Journal of Biomedical Informatics 47 (2014) 112–130. doi:10.1016/j.jbi.2013.10.001.

[4] S. Khan, S. Parkinson, Discovering and utilising expert knowledge from security event logs, Journal of Information Security and Applications 48 (2019) 102375. doi:https://doi.org/10.1016/j.jisa.2019.102375.

[5] J. Schmidt, M. R. G. Marques, S. Botti, M. A. L. Marques, Recent advances and applications of machine learning in solid-state materials science, npj Computational Materials 5 (1) (2019) 83. doi:10.1038/s41524-019-0221-0.

[6] I. H. Witten, E. Frank, M. A. Hall, C. J. Pal, Data Mining: Practical Machine Learning Tools and Techniques, 4th Edition, Morgan Kaufmann Publishers Inc., 2016.

[7] M. A. Munson, A study on the importance of and time spent on different modeling steps, SIGKDD Explor. Newsl. 13 (2) (2012) 65–71. doi:10.1145/2207243.2207253.

[8] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, I. H. , The weka data mining software: An update, SIGKDD Explorations Newsletter 11 (1) (2009) 10–18. doi:10.1145/1656274.1656278.

[9] M. R. Berthold, N. Cebron, F. Dill, T. R. Gabriel, T. Kötter, T. Meinl, P. Ohl, K. Thiel, B. Wiswedel, KNIME - the konstanz information miner: version 2.0 and beyond, SIGKDD Explorations 11 (1) (2009) 26–31. doi:10.1145/1656274.1656280.

[10] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, E. Duchesnay, Scikit-learn: Machine learning in Python, Journal of Machine Learning Research 12 (2011) 2825–2830.

[11] L. Beighley, Head first SQL, O'Reilly, 2007.

[12] R, The R Project for Statistical Computing, https://www.r-project.org/.

[13] W. McKinney, Data Structures for Statistical Computing in Python , in: Proceedings of the 9th Python in Science Conference, 2010, pp. 51 – 56.

[14] E. F. Codd, A relational model of data for large shared data banks, Commun. ACM 13 (6) (1970) 377–387. doi:10.1145/362384.362685. URL http://doi.acm.org/10.1145/362384.362685

[15] C. M. Wyss, E. L. Robertson, A formal characterization of PIVOT/UNPIVOT, in: O. Herzog, H. Schek, N. Fuhr, A. Chowdhury, W. Teiken (Eds.), Proceedings of the 2005 ACM CIKM International Conference on Information and Knowledge Management, Bremen, Germany, October 31 - November 5, 2005, ACM, 2005, pp. 602–608. doi:10.1145/1099554.1099709.

[16] C. Cunningham, G. Graefe, C. A. Galindo-Legaria, PIVOT and UNPIVOT: optimization and execution strategies in an RDBMS, in: M. A. Nascimento, M. T. Özsu, D. Kossmann, R. J. Miller, J. A. Blakeley, K. B. Schiefer (Eds.), Proceedings of the Thirtieth International Conference on Very Large Data Bases (VLDB), Morgan Kaufmann, 2004, pp. 998–1009. doi:10.1016/B978-012088469-8.50087-5.

[17] S. F. Crone, S. Lessmann, R. Stahlbock, The impact of preprocessing on data mining: An evaluation of classifier sensitivity in direct marketing, European Journal of Operational Research 173 (3) (2006) 781 – 800. doi:10.1016/j.ejor.2005.07.023.

[18] A. de la Vega, D. García-Saiz, M. E. Zorrilla, P. Sánchez, Lavoisier: High-level selection and preparation of data for analysis, in: Model and Data Engineering - 9th International Conference, MEDI 2019, Toulouse,

France, October 28-31, 2019, Proceedings, 2019, pp. 50–66. doi:10.1007/978-3-030-32065-2_4.

[19] U. Fayyad, G. Piatetsky-Shapiro, P. Smyth, From data mining to knowledge discovery in databases, AI magazine 17 (3) (1996) 37.

[20] T. Hartmann, A. Moawad, F. Fouquet, Y. L. Traon, The Next Evolution of MDE: a Seamless Integration of Machine Learning into Domain Modeling, Software & Systems Modeling 18 (2) (2019) 1285–1304. doi:10.1007/s10270-017-0600-2.

[21] P. Schlesinger, N. Rahman, Self-Service Business Intelligence Resulting in Disruptive Technology, Journal of Computer Information Systems 56 (1) (2016) 11–21. doi:10.1080/08874417.2015.11645796.

[22] M. Ester, H.-P. Kriegel, J. Sander, X. Xu, A density-based algorithm for discovering clusters a density-based algorithm for discovering clusters in large spatial databases with noise, in: Proceedings of the Second International Conference on Knowledge Discovery and Data Mining, KDD'96, AAAI Press, 1996, pp. 226–231.

[23] J. MacQueen, Some Methods for Classification and Analysis of Multivariate Observations, in: Proceedings of the Fifth Berkeley Symposium on Mathematical Statistics and Probability, Volume 1: Statistics, University of California Press, Berkeley, Calif., 1967, pp. 281–297.

[24] M. Mernik, J. Heering, A. M. Sloane, When and how to develop domain-specific languages, ACM Computing Surveys 37 (4) (2005) 316–344. doi:10.1145/1118890.1118892.

[25] E. Evans, Domain-driven design: tackling complexity in the heart of software, Addison-Wesley Professional, 2004.

[26] A. de la Vega, D. García-Saiz, M. Zorrilla, P. Sánchez, On the Automated Transformation of Domain Models into Tabular Datasets, ER FORUM 1979.

[27] A. de la Vega, Domain-Specific Languages for Data Mining Democratisation., PhD Thesis, Universidad de Cantabria, Spain (2019).
URL http://hdl.handle.net/10902/16728

[28] M. Fowler, Patterns of Enterprise Application Architecture, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[29] A. Kleppe, Software Language Engineering: Creating Domain-Specific Languages Using Metamodels, Addison-Wesley Professional, 2008.

[30] R. F. Paige, D. S. Kolovos, F. A. C. Polack, A tutorial on metamodelling for grammar researchers, Science of Computer Programming 96 (P4) (2014) 396–416. doi:10.1016/j.scico.2014.05.007.

[31] D. Steinberg, F. Budinsky, M. Paternostro, E. Merks, EMF: clipse Modeling Framework, 2nd Edition, Addison-Wesley Professional, 2009.

[32] M. Eysholdt, H. Behrens, Xtext: Implement Your Language Faster Than the Quick and Dirty Way, in: Companion to the 25th Annual Conference on Object-Oriented Programming, Systems, Languages, and Applications (SPLASH/OOPSLA), 2010, pp. 307–309. doi:10.1145/1869542.1869625.

[33] T. Kosar, S. Gaberc, J. C. Carver, M. Mernik, Program comprehension of domain-specific and general-purpose languages: replication of a family of experiments using integrated development environments, Empirical Software Engineering 23 (5) (2018) 2734–2763. doi:10.1007/s10664-017-9593-2.

[34] A. Barisic, V. Amaral, M. Goulão, Usability driven DSL development with USE-ME, Computer Languages, Systems & Structures 51 (2018) 118–157. doi:10.1016/j.cl.2017.06.005.

[35] A. de la Vega, D. García-Saiz, M. Zorrilla, P. Sánchez, FLANDM: a development framework of domain-specific languages for data mining democrati-

sation, Computer Languages, Systems and Structures 54 (2018) 316–336. doi:10.1016/j.cl.2018.07.002.

[36] T. Kosar, S. Bohra, M. Mernik, Domain-Specific Languages: A Systematic Mapping Study, Information and Software Technology 71 (2016) 77–91. doi:10.1016/j.infsof.2015.11.001.

[37] J. Bezanson, S. Karpinski, V. B. Shah, A. Edelman, Julia: A Fast Dynamic Language for Technical Computing, arXiv e-prints,arXiv:1209.5145.

[38] M. Boullé, C. Charnay, N. Lachiche, A scalable robust and automatic propositionalization approach for bayesian classification of large mixed numerical and categorical data, Machine Learning 108 (2) (2019) 229–266. doi:10.1007/s10994-018-5746-9.

[39] A. J. Knobbe, M. De Haas, A. Siebes, Propositionalisation and Aggregates, Principles of Data Mining and Knowledge Discovery 2168 (2001) 277–288. doi:10.1007/3-540-44794-6_23.

[40] J. M. Kanter, K. Veeramachaneni, Deep feature synthesis: Towards automating data science endeavors, in: International Conference on Data Science and Advanced Analytics (DSAA), 2015, pp. 1–10. doi:10.1109/DSAA.2015.7344858.

[41] M. Samorani, Automatically generate a flat mining table with dataconda, in: 2015 IEEE International Conference on Data Mining Workshop (ICDMW), 2015, pp. 1644–1647. doi:10.1109/ICDMW.2015.100.

[42] S. Džeroski, Relational Data Mining, Springer US, 2010, pp. 887–911. doi:10.1007/978-0-387-09823-4_46.

[43] E. Cilia, N. Landwehr, A. Passerini, Relational feature mining with hierarchical multitask kfoil, Fundam. Inform. 113 (2) (2011) 151–177. doi:10.3233/FI-2011-604.

[44] G. Manjunath, M. N. Murty, D. Sitaram, Combining heterogeneous classifiers for relational databases, Pattern Recognition 46 (1) (2013) 317–324. doi:10.1016/j.patcog.2012.06.015.
URL https://doi.org/10.1016/j.patcog.2012.06.015

[45] C. Nica, A. Braud, F. L. Ber, Exploring heterogeneous sequential data on river networks with relational concept analysis, in: Graph-Based Representation and Reasoning - 23rd International Conference on Conceptual Structures, ICCS 2018, Edinburgh, UK, June 20-22, 2018, Proceedings, 2018, pp. 152–166. doi:10.1007/978-3-319-91379-7_12.

[46] C. Abreu Ferreira, J. Gama, V. Santos Costa, Contrasting logical sequences in multi-relational learning, Progress in Artificial Intelligence 8 (4) (2019) 487–503. doi:10.1007/s13748-019-00188-w.

[47] M. A. Rodriguez, The gremlin graph traversal machine and language, CoRR abs/1508.03843. arXiv:1508.03843.

[48] R. Angles, J. Reutter, H. Voigt, Graph Query Languages, Springer, 2018, pp. 1–8. doi:10.1007/978-3-319-63962-8_75-1.

[49] E. F. Kendall, D. L. McGuinness, Ontology Engineering, Morgan & Claypool, 2019.

[50] P. P. Chen, The Entity-Relationship Model - Toward a Unified View of Data, ACM Transactions on Database Systems 1 (1) (1976) 9–36. doi:http://doi.acm.org/10.1145/320434.320440.

[51] R. Kimball, M. Ross, The Data Warehouse Toolkit, Wiley, 2013.

Alfonso de la Vega: Conceptualization, Methodology, Software, Validation, Writing - Original Draft, Writing - Review & Editing, Visualization
Diego García-Saiz: Conceptualization, Writing - Review & Editing
Marta Zorrilla: Writing - Review & Editing
Pablo Sánchez: Conceptualization, Methodology, Writing - Original Draft, Writing - Review & Editing, Supervision