IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received October 16, 2020, accepted October 28, 2020, date of publication November 6, 2020, date of current version November 18, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.3036462

Model-Driven Development of Web APIs to Access Integrated Tabular Open Data

CÉSAR GONZÁLEZ-MORA[1], DAVID TOMÁS[1], IRENE GARRIGÓS[1], JOSÉ JACOBO ZUBCOFF[2], AND JOSE-NORBERTO MAZÓN[1]

[1]Department of Software and Computing Systems, University of Alicante, 03690 Alicante, Spain
[2]Department of Sea Sciences and Applied Biology, University of Alicante, 03690 Alicante, Spain

Corresponding author: César González-Mora (cgmora@ua.es)

This work was supported by the National Foundation for Research, Technology and Development of the Spanish Ministry of Economy, Industry and Competitiveness under Project TIN2016-78103-C2-2-R and Project RTI2018-094653-B-C22. The work of César González-Mora was supported by a contract for predoctoral training with the Generalitat Valenciana and the European Social Fund under Grant ACIF/2019/044.

ABSTRACT More and more governments around the world are publishing tabular open data, mainly in formats such as CSV or XLS(X). These datasets are mostly individually published, i.e. each publisher exposes its data on the Web without considering potential relationships with other datasets (from its own or from other publishers). As a result, reusing several open datasets together is not a trivial task, thus requiring mechanisms that allow data consumers (as software developers or data scientists) to integrate and access tabular open data published on the Web. In this paper, we propose a model-driven approach to automatically generate Web APIs that homogeneously access multiple integrated tabular open datasets. This work focuses on data that can be integrated by means of join and union operations. As a first step, our approach detects unionable and joinable tabular open data by using a table similarity measure based on word embeddings. Then, an APIfication process is developed to create APIs that access the previously integrated datasets through a single endpoint. A running example is presented throughout the article, as well as a set of experiments for performance evaluation to show the feasibility of our approach.

INDEX TERMS Data integration, join, union, open data, data access, Web APIs, word embeddings.

I. INTRODUCTION

Nowadays the amount of open data available on the Web is increasing due to the great interest of governments and institutions around the world in adopting open data initiatives [1]. A good example is found in the smart city arena, where open data has attracted great attention for local and regional governments as the best way of publishing the big data they are producing [2].

This open data is commonly offered in catalogues within Web platforms, named *open data portals*. For instance, *https://www.data.gov/*, which provides a catalogue of data resources from the USA Government (at national level), or *https://data.cityofchicago.org/*, which gathers open data from Chicago (at local level).

The ultimate goal of open data portals is to provide Linked Open Data (LOD) that allows consumers to use Semantic Web technologies to identify relationships among data [3]. LOD is easy to integrate and access by means of query languages such as SPARQL. Unfortunately, LOD is not usually available since most used formats in open data government portals are tabular (46.4%), either direct tabular formats such as CSV (9.1%) and XLS(X) (6.1%), or embedded tabular data such as HTML (25.0%) and PDF (9.2%). Meanwhile, LOD formats such as RDF only represent 0.5% of the total [4] (even though there exist proposals to transform from CSV to RDF [5]). The prevalence and priority of tabular formats over LOD in open data government portals has been highlighted in recent studies by the European Comission [6] and the Organisation for Economic Co-operation and Development (OECD) [7]. In this scenario, accessing and manipulating tabular datasets remains a relevant research topic in the field of open data.

When it comes to integrating tabular open data, additional metadata is required for developers to know how datasets can be related to each other (such as relational-like primary keys

The associate editor coordinating the review of this manuscript and approving it for publication was Porfirio Tramontana.

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

or foreign keys, as stated by the "Model for Tabular Data and Metadata on the Web"[1] developed by the W3C CSV on the Web Working Group). Although several recent approaches proposed adding these kind of metadata to tabular datasets by means of annotations [8], [9], this methodology has not being widely adopted by publishers. Therefore, open data is mostly individually published, i.e. each publisher shares their data on the Web without considering potential relationships with other open data. This scenario hampers users of open data government portals (such as software developers or data scientists) in reusing open data, since additional effort must be done to successfully integrate this data. Thus, mechanisms that allow data consumers to integrate and access tabular open data published on the Web (through open data government portals) are highly required.

To this end, in this paper we propose a model-driven APIfication process to define transformations for automatically generating Web APIs that access integrated tabular open data. Data integration is a complex process, involving several kinds of transformations (such as cleansing, combination, normalisation, etc.) to offer a unified view of a set of heterogeneous data from different sources. In this paper, we focus on join and union operators since they are specially relevant for open data integration [10]. In addition, it should be noted that considering these operators provides the basis for including other ones (such as filtering and sorting).

Although data integration challenges have been researched for years with significant progress [11], efforts have been made only on solving specific problems. However, according to Abadi *et al.* [12] more work is required on researching how to pipeline data integration to cover all the way from raw data to an end-user's desired outcome. This could be achieved, for instance, by means of generating mechanisms that support developers and data scientists in consuming the right data for their purposes by using external programming languages such as Java, Python, and R. In this sense our work is aligned with Abadi *et al.*, since the approach we propose focuses on pipelining the data integration process together with an API generation in order to simplify consumption of integrated open data.

The first step in the approach proposed consists of detecting the tabular datasets that are more likely to be integrated by means of join and union operations.[2] For this purpose, we defined a similarity measure between tabular data based on word embeddings [13]. Afterwards, in a second step a Web API is automatically generated to directly access the integrated datasets. Web APIs are a recommended feature of open data portals [14], allowing data consumers to build data-intensive applications and bring open data to citizens. Moreover, the documentation of the Web API is also automatically generated (i.e. its interactive OpenAPI[3] documentation), helping data consumers to better understand how to access and reuse integrated tabular open data coming from different sources.

It is worth noting that both steps in the APIfication process are decoupled and can be used independently, while at the same time relationships among steps (e.g. passing information from integrated data to a data model in order to initiate the Web API generation) can be easily considered by following a model-driven development approach.

In summary, the contributions of this article are as follows:

- The definition of a word embedding-based similarity measure between tabular datasets to identify joinable and unionable tabular open data.
- A set of model-driven transformations to automatically generate a Web API to easily access previously integrated data (by applying the corresponding join and union operations).
- Evaluation of our approach with real tabular open data.
- The implementation of our approach for automatically data integration and the corresponding Web API generation, which is available online at GitHub.[4]

This article is structured as follows. Section II presents the running example used to illustrate the approach explained in Section III. That section describes our model-driven approach for automatically generating Web APIs to access integrated tabular open data. Then, Section IV presents the evaluation of the approach. Finally, related work is described in Section V and conclusions and future work are sketched out in Section VI.

II. RUNNING EXAMPLE

This section introduces a running example which is used throughout the article to illustrate our approach. The considered scenario is related to available open data regarding the COVID-19 pandemic coming from different open data portals, and how they can be integrated and accessed through an automatically generated Web API. All the files related to the running example are publicly available online.[5]

This running example describes a situation where a data scientist is willing to use available open data to create a dashboard to analyse the COVID-19 pandemic evolution along different cities in USA. However, the data scientist has to address a data integration problem, since open data about COVID-19 is published at different portals, by different local or regional governments (USA cities or states), and usually federated in a national open data portal. This data may rely on different schemas (i.e. data structures), being necessary to integrate them first in order to successfully analyse them together.

In our running example, data comes from USA Government's open data portal.[6] On one hand, data from Chicago is

[1] https://www.w3.org/TR/tabular-data-model/

[2] It is worth noting that our approach could be used for any tabular format, although we focus on CSV files in this article.

[3] https://www.openapis.org/

[4] https://github.com/cgmora12/DataIntegration2API

[5] https://github.com/cgmora12/DataIntegration2API/tree/master/runningExample

[6] https://www.data.gov/

gathered from two different datasets. The first one[7] provides figures of positive COVID-19 cases by day, filtering by gender, race, and different ranges of age. It contains 148 rows and 39 columns. An excerpt of this data (called *dataset 1*) is shown in Table 1. The second one[8] (*dataset 2*) provides hospital capacity metrics during COVID-19 period. It contains 132 rows and 33 columns. See Table 2 for a sample data.

TABLE 1. Excerpt of data from COVID-19 cases in Chicago (*dataset 1*).

date	cases - total	deaths - total	cases - age 0-17	cases - male	cases - female
03/25/2020	367	5	2	196	170
03/26/2020	416	2	2	230	184
03/27/2020	404	8	4	192	210

TABLE 2. Excerpt of hospital capacity data from Chicago (*dataset 2*).

date	ventilators total capacity	ventilators in use COVID-19 patients	ventilators in use non-COVID-19
03/25/2020	1,048	79	343
03/26/2020	1,053	88	295
03/27/2020	1,042	108	309

On the other hand, open data from New York City is gathered from a unique dataset[9] (*dataset 3*) that includes daily counts of cases, hospitalisations, and deaths from COVID-19. It contains 151 rows and 4 columns. An excerpt is shown in Table 3.

TABLE 3. Excerpt of data from COVID-19 cases in New York City (dataset 3).

date	case count	hospitalization count	death count
2020 Mar 22 12:00:00 AM	2580	725	50
2020 Mar 23 12:00:00 AM	3568	1037	82
2020 Mar 24 12:00:00 AM	4504	1153	94

In order to consider data provenance, we automatically add a new column to each dataset containing its publisher before processing the data: in dataset 1 and dataset 2 the publisher is ''City of Chicago'', whereas ''City of New York'' is the publisher of dataset 3. This provenance data comes from the datasets' metadata (USA Government's open data portal) and can be easily obtained through its API.

[7]https://catalog.data.gov/dataset/covid-19-daily-cases-and-deaths, last accessed October 2020

[8]https://catalog.data.gov/dataset/covid-19-hospital-capacity-metrics, last accessed October 2020

[9]https://catalog.data.gov/dataset/covid-19-daily-counts-of-cases-hospitalizations-and-deaths, last accessed October 2020

If a data scientist wants to use this data to get the COVID-19 positive cases, deaths, and people hospitalised by day using ventilators, three datasets must be downloaded and integrated by applying join and union operators (a more detailed explanation about these operators is provided in the next section) as follows (see also Fig. 1):

- Dataset 1 and dataset 2 contain data with different measures that can be joined by the column ''date'' (resulting in a new dataset) in order to have positive cases, deaths, and hospitalised people using ventilators by day in Chicago. For example, the 25th of March, there were 367 cases and 5 deaths in Chicago (dataset 1), and also 79 hospitalised people that used ventilators due to COVID-19 (dataset 2). All this information is joined in the same row because they share the same ''date'', which is used as a condition for the join operator.
- Union of the previously joined dataset and dataset 3 can be applied to get the positive cases, deaths, and hospitalised persons using ventilators by day in both Chicago and New York. For example, in order to perform the union operation, the columns are matched as follows: ''date'' with ''date'' (previously considering that a date format conversion must be done), ''case count'' with ''cases'', ''death count'' with ''deaths'' and ''hospitalized count'' with ''ventilators''. Matching these columns and applying a union operator results in a new table containing all the rows from the input tables. Fig. 1 shows that the first row of the integrated data comes from dataset 3, whereas the second row comes from the previously joined datasets. Additionally, a new column has been included with information about the publisher of the data.

Manually obtaining the integrated dataset required in this example is a time-consuming task for the data scientist. In the following section, we use this running example to show how our approach can be useful to save time and effort in integrating and accessing tabular open data.

III. A MODEL-DRIVEN APIfication APPROACH TO ACCESS INTEGRATED TABULAR OPEN DATA

This section presents our model-driven APIfication approach to automatically generate Web APIs to access previously integrated tabular open data.

Operators considered for handling input tabular data are union and join from relational algebra. For the sake of simplicity, in this article we borrow these operators from SQL (the well-known implementation of the relational algebra and a recognised standard for querying and handling tabular data):

- Union operator is denoted by $\cup$ symbol in relational algebra. Given two tabular datasets (A and B), union operator aims to get a unique dataset that contains rows that are in A or in B (denoted as $A \cup B$). A and B must have the same columns (number, order, and datatype) to be computed. Also, each column of each dataset must refer to the same concept to be meaningful.

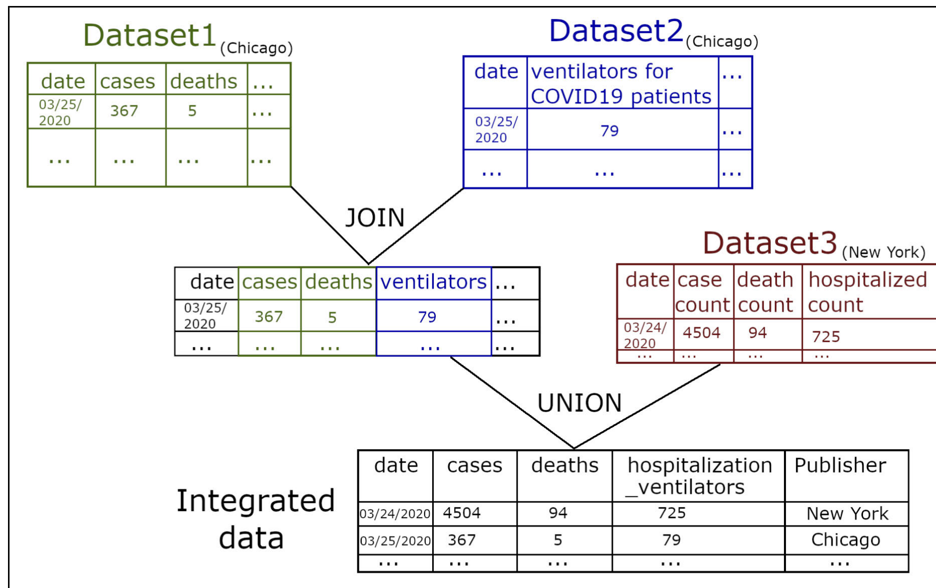

FIGURE 1. Example of integrating COVID-19 datasets.

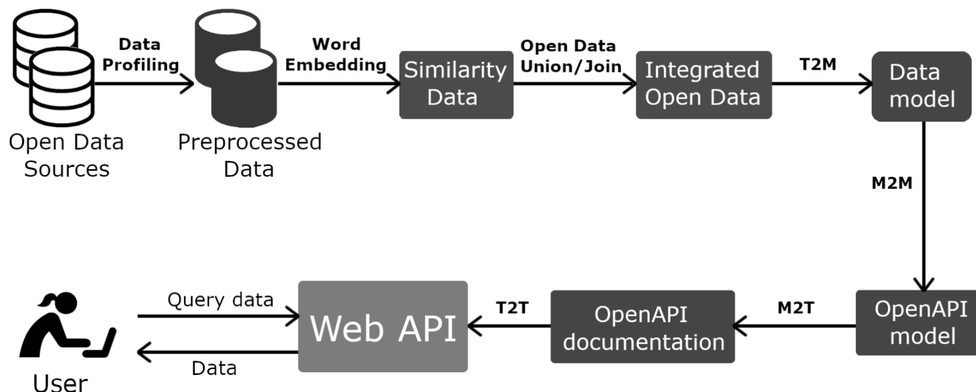

FIGURE 2. Automatic open data integration and API generation process.

- Join operator is denoted by ⋈ symbol in relational algebra. Given two tabular datasets A and B, join operator aims to get a unique dataset that includes every column from A and B (denoted as A ⋈ B) and contains rows that fulfil a matching condition (applied to values of some columns).

An overview of our approach is shown in Fig. 2. It consists of two parts:

- Data integration (“Similarity Data” and “Integrated Open Data” boxes in Fig. 2). The first step implies detecting unionable and joinable tabular datasets from an open data portal. Our approach uses word embeddings [13] to detect which columns from different input tabular datasets are more likely to be integrated by using join and union operators. The next step consists of applying the required join and union operators to get an integrated tabular dataset. More details are given in Section III-A. As shown in Fig. 2, prior to this data integration phase a data profiling is performed in order to discover data types and to apply type conversions to homogenise them (e.g. conversion of date formats by using the “mm/dd/yyyy” pattern), as well as to detect keys that will be used later to apply the join operator.
- Model-driven transformations to obtain a Web API to access integrated data, including its interactive OpenAPI documentation: “Data model”, “OpenAPI model”, “OpenAPI documentation”, and “Web API” boxes in Fig. 2. From the previously integrated dataset, a Web API to access this data is generated by means of the following model-driven transformations: (i) a text to model (T2M) transformation from the integrated data to the data model; (ii) a model to model (M2M) transformation

from the data model to the OpenAPI model; (iii) a model to text (M2T) transformation from the OpenAPI model to its OpenAPI documentation; and finally (iv) a text to text (T2T) transformation from the OpenAPI documentation to the Web API. Therefore, starting from a dataset containing rows and cells, the system performs a direct transformation to construct a data model object with row and cell objects. Then, the data model is transformed to an OpenAPI model using each cell of the first row as an API component (method, parameter, and property), parsing this OpenAPI model object to a standard format for API documentation. Finally, the complete Web API is automatically created according to the methods detailed in its documentation. These model-driven transformations are explained in detail in Section III-B.

A. INTEGRATION OF TABULAR OPEN DATA

As mentioned before, the first part of the process consists of integrating datasets by means of join and union operators. The initial step is to determine which columns of the datasets are similar enough to apply these operations. The following sections explain in detail how similar columns are detected, as well as how union and join operators are applied for integrating tabular data (we specifically focus on CSV files).

1) MEASURING COLUMN SIMILARITY FOR DETECTING UNIONABLE AND JOINABLE TABULAR DATA

Word embeddings is a Natural Language Processing (NLP) technique in which words or phrases are mapped to vectors of real numbers. This mapping is based on the distributional hypothesis, which states that words that occur in the same contexts tend to have similar meanings [15]. Word embeddings have been shown to capture semantic regularities in vector space, since the relationship between two word vectors mirrors the linguistic relationship between those words [13].

In order to determine whether union or join operations between tabular data can be performed, we have established a mechanism based on word embeddings to calculate the (semantic) similarity of two tabular datasets. Using word embeddings overcomes the problems of lexical approaches based on string similarity: terms such as ''city'' and ''location'' could be considered as being very different in terms of string matching, but in a word embedding space these two terms may be closely related and considered as highly similar. The similarity mechanism takes as an input a set of tabular datasets to be compared, returning as a result a JSON file where all the columns of each dataset are compared with each other, obtaining a similarity measure for each pair of columns belonging to different datasets. These column pairs are sorted in descending order of similarity. This measure serves as the input to decide whether union or join operators can be applied to integrate different datasets (see Section III-A2 for additional information). In order to calculate the similarity between columns, we take into account two elements of the tabular datasets: name of the columns and their content (values) for each row.

The first step consists of normalising these values by splitting CamelCase and hyphenated words (very common in column names), removing punctuation, and converting text to lowercase. After that, the word embedding model is used to extract two vectors for each column: one represents the name of the column and the other the content of the column for each row of data. In those situations where the name of the column includes more than one word, the vectors representing each word are averaged to get a single vector. Averaging word embeddings is one of the most popular methods of combining embedding vectors, outperforming more complex techniques especially in out-of-domain scenarios [16]. The same strategy is applied to the content of the column, where the final vector is the result of calculating the mean between the vectors of each of the values contained.

As in previous works [13], we use the cosine similarity to compute the distance between vectors in the embedding space:

$$sim(v_1, v_2) = \frac{v_1 v_2}{\|v_1\| \|v_2\|} = \frac{\sum_{i=1}^{n} v_{1i} v_{2i}}{\sqrt{\sum_{i=1}^{n} (v_{1i})^2} \sqrt{\sum_{i=1}^{n} (v_{2i})^2}},$$

where v_1 and v_2 represent the word embedding vectors of the name of the columns or the content of the column for each row, and $sim(v_1, v_2)$ is a float value in the range [0, 1], where 0 means no similarity and 1 means maximum similarity between the vectors considered.

If the word embedding model does not provide coverage for the name of the column or its content (i.e. their tokens are not in the vocabulary of the model), the Levenshtein distance [17] is used as a backup strategy to ensure that the system always returns a similarity value between columns. This string metric is based on the number of single-character edits (insertions, deletions or substitutions) required to change one string into the other. We applied the normalised edit distance to obtain values in the range [0, 1], computed as $(length - distance)/length$, where *distance* is the Levenshtein distance and *length* is the sum of the lengths of the two strings to compare.

For each two columns compared, we obtain a similarity value of the name of the column and a similarity value of its content. To obtain a single final similarity score of two columns c_1 and c_2, we perform a linear combination of these two values:

$$sim(c_1, c_2) = \alpha \cdot sim(c_{n1}, c_{n2}) + (1 - \alpha) \cdot sim(c_{c1}, c_{c2}),$$

where $sim(c_{n1}, c_{n2})$ is the similarity of the names of the columns, $sim(c_{c1}, c_{c2})$ is the similarity of their contents, and α is a parameter in the range [0, 1] that weights the relevance of the two similarity scores in the final result.

2) APPLYING UNION AND JOIN OPERATIONS FOR DATA INTEGRATION

The similarity measure computed in the previous step is used to decide whether union or join operations can be applied for integrating tabular data. Two rules apply:

- A join operator can be applied to datasets that have at least one column (detected as candidate key in the data profiling) with similarity value higher than a specific threshold (as explained below). These columns are considered as key columns in the join operation (several columns can be detected as key for joining).
- A union operator can be applied to datasets that have all columns with similarity value higher than a specific threshold. Columns with similarity below this threshold can be removed previously. Also, if the majority of columns are similar, the union operation can be applied only on these columns omitting the rest of non-similar columns.

The value of the thresholds mentioned are obtained empirically considering that: (i) join operation requires key columns, which are a small subset of columns from the input datasets (usually one or two); (ii) union operation requires the same columns to be present in both input datasets. In the running example, the threshold for join operator was set to 0.95, whereas for union operator was set to 0.75. However, our approach is flexible and thresholds can be adapted at any time.

Join and union operators between datasets are automatically applied by dynamically developing and executing transformations with Pentaho Data Integration Java libraries.[10] Therefore, after analysing the similarity calculations we can automatically create a transformation using these libraries and execute it to obtain the integrated data. This generated transformation includes the following operations: (i) reading the datasets to be integrated; (ii) sorting data; (iii) computing similarity measures to apply join/union operations; and (iv) save the integrated data to an output file. An excerpt of the code required is shown in Fig. 3 (the full code is available at the GitHub repository[11]).

Regarding the running example, a join operation is performed automatically on the two datasets from Chicago (dataset 1 and dataset 2) because the similarity of "date" column from COVID-19 cases dataset and the "date" column from Hospital Capacity Metrics is the highest computed (0.9988) and above the threshold of 0.95. Both "date" columns are selected as key columns for the join operator. After joining, we obtain a single tabular dataset with data coming from both datasets (see Table 4). Moreover, after calculating the similarity between columns from the previously integrated datasets and dataset 3 from New York, the similarity obtained is higher than the 0.75 threshold for the following matching columns: "case count" with "cases total" (0.8847), "death count" with "deaths total" (0.8724), "date of interest" with "date" (0.8607), and "hospitalized count" with "ventilators in use covid 19 patients" (0.7793).

```java
// Create transoformation
StepLoader.init();
TransMeta transMeta = new TransMeta();
transMeta.setName("Join Transformation");

// First step: read CSV
CsvInputMeta csvinput = new CsvInputMeta();
csvinput.setFilename(csvFilename1);
csvinput.setDelimiter(separator1);
csvinput.setHeaderPresent(true);
csvinput.setEnclosure("\"");
csvinput.setBufferSize("50000");
csvinput.setInputFields(configureInputFields());

StepMeta csvinputStepMeta = new StepMeta("CsvInput", csvinput);
csvinputStepMeta.setDraw(true);
csvinputStepMeta.setLocation(100, 100);
transMeta.addStep(csvinputStepMeta);
```

FIGURE 3. Excerpt of Java code for generating a join transformation.

TABLE 4. Excerpt of joined dataset from COVID-19 cases in Chicago.

date	cases - total	deaths - total	ventilators in use COVID-19 patients
03/25/2020	367	5	476
03/26/2020	416	2	88
03/27/2020	404	8	479

Therefore, a union operation is performed to integrate both datasets based on these matching columns.

Although a Pentaho Data Integration transformation is created and executed dynamically by using its corresponding Java libraries, the transformation for the running example can be accessed by using the Pentaho Data Integration GUI, named Spoon, as shown in Fig. 4. It is worth noting that, by doing this, Spoon capabilities for debugging transformations can be easily used if required. For example, users could use Spoon for manually editing joining keys automatically selected previously by our approach.

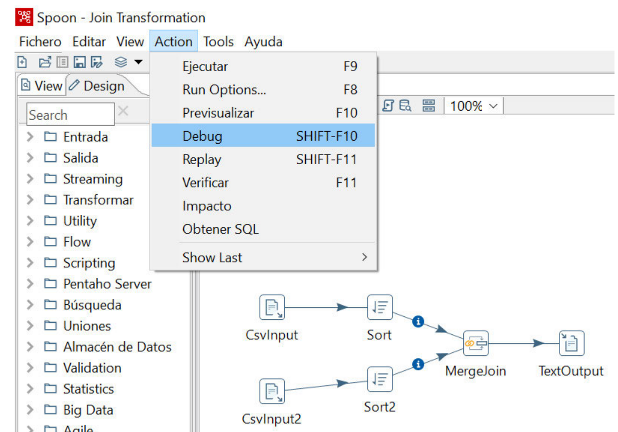

FIGURE 4. Example of join transformation auto-generated dynamically.

Once the data is integrated by applying join and union operators, a Web API is automatically generated to access these data (Table 5) as shown in the following section. This process of generating the Web API from the integrated data

[10]https://www.hitachivantara.com/en-us/products/data-management-analytics/pentaho-platform/pentaho-data-integration.html

[11]https://github.com/cgmora12/DataIntegration2API/blob/7cca3fb915f933a315858ac7ae6e0e1358f7df33/src/table/union/TableUnion2API.java#L341

TABLE 5. Excerpt of the dataset obtained after union operation of the datasets from COVID-19 cases in Chicago and New York.

date	cases - total	deaths - total	hospitalized (with ventilators)	publisher
03/22/2020	2580	50	725	City of New York
03/23/2020	3568	82	1037	City of New York
03/24/2020	4504	94	1153	City of New York
03/25/2020	367	5	476	City of Chicago
03/26/2020	416	2	295	City of Chicago
03/27/2020	404	8	479	City of Chicago

is decoupled from data integration itself and can be used independently. However, at the same time, the required information is easily transferred throughout the process thanks to the model-driven development principles. In particular, after obtaining the similarity results, the decision on which tables to perform the union/join operation depends on a threshold. Although there is a default threshold, users could change it at the beginning of the data integration process. This decision does not affect the process of API generation since only already integrated data, together with other information stored in the models, are required as an input.

B. MODEL-DRIVEN TRANSFORMATIONS FROM INTEGRATED DATA TO WEB API

After generating the integrated data, the transformation process continues in order to generate a Web API that offers easy access to this data. As shown in Fig. 2, this transformation from the integrated data to the Web API contains a text to model (T2M) transformation to create a data model, a model to model (M2M) transformation to generate the documentation model in OpenAPI, a model to text (M2T) transformation to obtain the complete OpenAPI documentation, and finally a text to text (T2T) transformation that completes the process by producing the Web API.

The T2M transformation starts from the integrated open data, generating the related data model. It consists of a MOF-based[12] model in XMI format according to its metamodel defined in Fig. 5 by using the Ecore format (actually, the *de facto* reference implementation of MOF) from the Eclipse Modeling Framework (EMF). This metamodel specifies that tabular data as CSV file contains a name and a set of rows, which include a position and a set of cells with value and type. The first row of a tabular dataset, which represents the column names, is used for the API method, properties, and parameters; the second row, which contains the data, is used as example values for each column. Therefore, the integrated data in a CSV file is processed by rows, analysing the first two rows for creating the data model with row and cell objects.

Particularly, in the running example the generated data model shown in Fig. 6 contains an object "Table", formed by a set of "Rows" containing many "Cells". The information

[12]https://www.omg.org/mof/

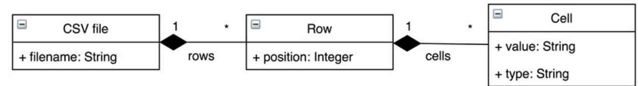

FIGURE 5. Tabular dataset metamodel.

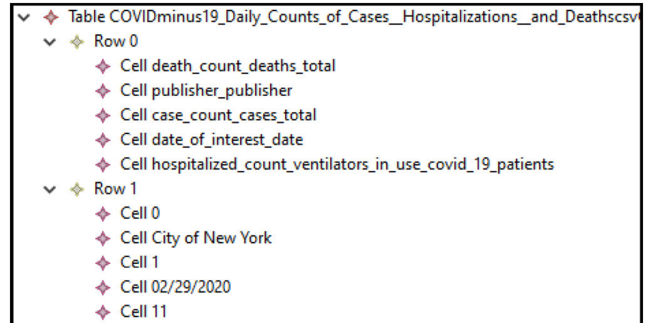

FIGURE 6. Datafile model in XMI format.

contained in the cells of the first row consists of the column names, while the cells in the second row contain integrated data examples about COVID-19. An example of column name is "date_of_interest_date", which is converted into a cell in the first row (Row 0 in Fig. 6). This cell, and all the cells from the first row, will then be used as the name of an API method, parameter, and property in the subsequent transformation steps. On the other hand, an example value of "date_of_interest_date" is "02/29/2020", which is converted into a cell in the second row (Row 1 in Fig. 6) and will be used as an example value in the corresponding API method, parameter, and property.

Once the data model is created, the M2M transformation is performed. In this step an OpenAPI model is created from the data model. This transformation is defined using ATL, one of the most used languages for model transformations [18], [19]. Using the ATL language we defined a set of transformation rules (see Fig. 7) that automatically convert the data model

```
-- @atlcompiler emftvm
-- @path Table=/TransformationRules/Table.ecore
-- @path Openapi=/TransformationRules/Openapi.ecore
module Table2Openapi;

create OUT: Openapi from IN: Table;
rule Main {
    from
        s: Table!Table
    using {
            firstTableRow : Sequence(Table!Cell) = s.rows->first().cells;
            lastTableRow : Sequence(Table!Cell) = s.rows->last().cells;
    }
    to
        t: Openapi!API (
            openapi <- '3.0.0',
            info <- openapi_info,
            servers <- Sequence{openapi_servers},
            paths <- Sequence{openapi_basic_path}.union
                (firstTableRow -> collect(e | thisModule.OpenapiPaths(e, s))),
            components <- Sequence{openapi_components}
        ),
        openapi_info: Openapi!Info (
            title <- s.filename,
            version <- '1.0.0',
            description <- 'Obtaining the ' + s.filename
        ),
        openapi_servers: Openapi!Server (
            url <- 'http://www.urlprueba.com/v1'
        ),
```

FIGURE 7. Excerpt of ATL transformation rules.

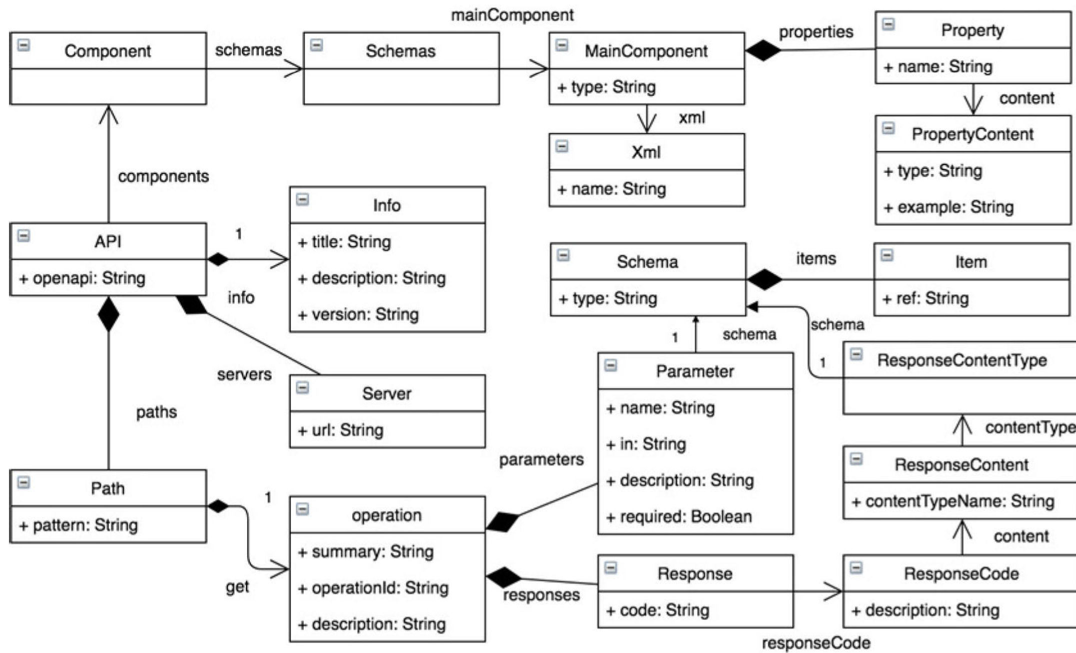

FIGURE 8. OpenAPI metamodel.

to the OpenAPI model. Starting with the table object from the data model, we are able to generate the OpenAPI model and all the different objects contained in its OpenAPI metamodel, which has been created by the authors according to the OpenAPI structure (Fig. 8). The main elements of this OpenAPI metamodel are: the `API` object containing OpenAPI information about the version and the URL of the Web API; the `Path` object which includes a set of API operations with their parameters; and the `Component` object to define the properties of the API.

The generated OpenAPI model for the running example is shown in Fig. 9. The "date_of_interest_date" column name from the running example is converted into a "Path" object with the pattern "/date_of_interest_date/ date_of_interest_date" including a GET operation, a "Parameter" object "date_of_interest_date" that can be used in this operation, and a "Property" named "date_of_interest_date".

After that, the M2T transformation between the OpenAPI model and the OpenAPI documentation is carried out. This OpenAPI documentation of the Web API is represented by a JSON file according to the standards of Swagger.[13] This JSON file is directly inferred from the OpenAPI model in XMI format by a simple element to element transformation, since the OpenAPI model contains the same elements than the API documentation. An excerpt of the OpenAPI documentation in JSON format generated for the running example is shown in Fig. 10. It contains important elements such as API information, server URL, components of the API, and paths to obtain the COVID-19 data filtering by different parameters and properties. The "Path" object "date_of_interest_date"

```
API 3.0.0
  Server http://localhost:8080
  Path /
  Path /date_of_interest_date/{date_of_interest_date}
    Operation GET date_of_interest_date
      Parameter limit
      Parameter offset
      Parameter date_of_interest_date
      Response 200
  Path /death_count_deaths_total/{death_count_deaths_total}
  Path /publisher_publisher/{publisher_publisher}
  Path /case_count_cases_total/{case_count_cases_total}
  Path /hospitalized_count_ventilators_in_use_covid_19_patients/
  Component
```

FIGURE 9. Excerpt of the OpenAPI model (XMI).

from the OpenAPI model is now converted into a JSON object inside the "Path" JSON array. This "date_of_interest_date" JSON object also contains more details about the GET operation and the parameter to filter the information.

Finally, the complete Web API is generated by the T2T transformation from the OpenAPI documentation. The T2T process is able to automatically generate the whole Web API represented by a server in NodeJS.[14] This API generation is performed with the help of the Swagger Codegen tool,[15] which helps in the creation of the structure required for the API, managing the calls to its different methods. Next, the code for the methods of the API is provided by our approach, which includes the required features to retrieve and

[13] https://swagger.io

[14] https://nodejs.org

[15] https://swagger.io/tools/swagger-codegen/

```
▶ components {1}
▼ servers [1]
  ▼ 0 {1}
      url : https://wake.dlsi.ua.es/IntegrationAPI
  openapi : 3.0.0
▼ paths {7}
  ▶ /visualisation {1}
  ▶ /case_count_cases_total/{case_count_cases_total} {1}
  ▶ /date_of_interest_date/{date_of_interest_date} {1}
  ▶ /death_count_deaths_total/{death_count_deaths_total} {1}
  ▶ /hospitalized_count_ventilators_in_use_covid_19_patients/

  ▶ /publisher_publisher/{publisher_publisher} {1}
  ▶ /     {1}
▼ info {3}
    description : Obtaining the
                  COVIDminus19_Daily_Counts_of_Cases__Hospital
                  scsvCOVIDminus19_Daily_Cases_and_Deathscsvcs
    title : COVIDminus19_Daily_Counts_of_Cases__Hospitalizatio
            VIDminus19_Daily_Cases_and_Deathscsvcsv
    version : 1.0.0
```

FIGURE 10. Excerpt of the OpenAPI JSON file.

return the integrated data requested by the data consumers. To this end, we first developed a base programming code which is then automatically added to each generated API. This code is then adapted to the method of the API and the source dataset, being able to perform the operations to read the dataset, search and filter specific information, and return it to the users.

There is a direct relationship between the OpenAPI and the API to generate: each "Path" specified in the OpenAPI documentation will result in a different method of the API, and each parameter will be used by the API for filtering and returning the results. The structure of the API also contains: an "api" folder, which includes a Swagger file defining the structure of the API; the folder "controllers", which includes the main controller to manage the queries and redirect them to the default controller to execute the query, get the information and return it to the user; a "node_modules" folder with the required libraries to implement the NodeJS server; the file "data.csv" containing the CSV input data; and a set of additional files including API information.

The generated Web API can be published in an online server so that users can query and filter the integrated data with the desired parameters. Data consumers are also able to perform API queries using the interactive OpenAPI documentation. This interactive documentation reduces the possibility of introducing errors such as writing mistakes. For example, if data consumers want to query the API directly and filter the data by a parameter called `case_count_cases_total`, they have to specify the exact parameter taking into account symbols and lowercase. However, with our interactive documentation data consumers do not need to write down the parameter since they can just click on the specific method to filter by the desired parameter.

With regard to the running example, the OpenAPI model generated contains all the objects with the specific information. That is, the different paths and components of the API to retrieve the integrated COVID-19 data filtering by different parameters, which are indeed the columns contained in the integrated data. The API of the running example is available at https://wake.dlsi.ua.es/IntegrationAPI/, whereas the corresponding documentation (see Fig. 11) is at https://wake.dlsi.ua.es/IntegrationAPI/docs/.

When the API from the running example is queried, the COVID-19 data that fulfils the specified parameters (column values) is retrieved. For instance, a query that requests the COVID-19 data in date 03/25/2020 is: *https://wake.dlsi.ua.es/IntegrationAPI/?date_of_interest_date=03/25/2020*. From this request, the Web API returns the required information in JSON format. The result obtained from this query example is the data about COVID-19 in the date specified, containing the following information:

```
{
 "date_of_interest_date": "03/25/2020",
 "publisher_publisher":
    "City of New York",
 "death_count_deaths_total": "123",
 "case_count_cases_total": "4864",
 "hospitalized_count_ventilators_
    in_use_covid_19_patients": "1303"
},
{
 "date_of_interest_date": "03/25/2020",
 "publisher_publisher":
    "City of Chicago",
 "death_count_deaths_total": "5",
 "case_count_cases_total": "368",
 "hospitalized_count_ventilators_
    in_use_covid_19_patients": "79"

 }
```

IV. EVALUATION

This section describes the evaluation carried out on our approach. We evaluated three aspects of the system: the word embeddings-based similarity approach for tabular data integration, the automatic generation of Web APIs to access integrated data, and the execution time performance of the entire pipeline.

A. WORD-EMBEDDINGS FOR DISCOVERING JOINABLE AND UNIONABLE TABLES

This section presents an intrinsic evaluation of the word embeddings-based similarity approach for tabular dataset integration. The goal of these experiments is not only to evaluate the performance of the similarity approach, but also to identify the best model and parameters to be used in the subsequent API generation stage (described in Section IV-B).

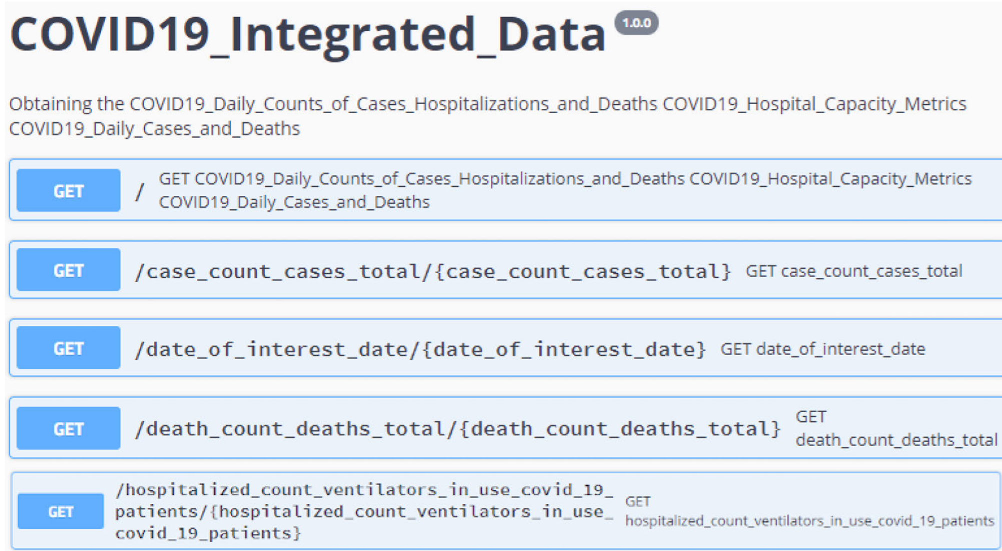

FIGURE 11. Interactive OpenAPI documentation of the API.

1) EXPERIMENTAL SETUP

In the evaluation carried out, we assimilated the task of computing similarity of tabular datasets to the task of *ad hoc* table retrieval: answering a search query with a ranked list of tables [20]. In this context, the search query is not a sequence of keywords but a table itself [21].

Given a set of tables T, the similarity $sim(t_1, t_2)$ for every pair $t_1, t_2 \in T$ is computed as:

$$sim(t_1, t_2) = \frac{\sum_{i=1,j=1}^{i \leq n, j \leq m} sim(c_{1i}, c_{2j})}{|C_1||C_2|},$$

where $C_1 = \{c_{11}, c_{12} \ldots, c_{1n}\}$ and $C_2 = \{c_{21}, c_{22} \ldots, c_{2m}\}$ are the set of columns of t_1 and t_2 respectively. Thus, the similarity between two tables is computed as the average similarity between their columns (calculated using the formula described in Section III-A1). For a given table, the system will return a ranked list of related tables based on this similarity measure.

The dataset used to test the approach was developed by Nargesian *et al.* [22] and it is publicly available for the evaluation of approaches related to tabular data integration. This database consists of more than 5,000 tables in CSV format extracted from USA, Canada, and UK open data portals, providing a ground truth that identifies which columns of a table match the columns of the other tables. The dataset was built starting with 32 base tables, which were manually aligned to identify matching columns. The final set was created by first issuing a projection on a random subset of columns of a base table, and then a selection with some limit and offset on the projected table. The tables contain the name of the columns and the corresponding content of the cells, comprising text, numeric, and date values.

Although the word embedding approach is specially suitable for textual data, our proposal also provides coverage for columns containing other data types. First, the name of the columns are textual data, even if its contents are numbers or dates. Thus, the system can return a similarity value based solely on the column names. Secondly, if the content of the columns is considered, even though all the possible numeric values are not represented in the embedding space, the word embedding models still provide coverage for many of them. For instance, the fastText model described below covers 99.90% of the numbers ranging from 0 to 10,000. This implies that any numeric value used to represent days, months, or years has a vector representation in the model.

In order to perform the experiments, a subset of 1,000 tables was randomly selected. Every table in this subset was used as a query to the system and compared with all the other tables in it, obtaining a ranked list of the most similar tables according to the measure given above. In our experiments, we consider a returned table to be relevant to a query table if there is at least one matching column between them in the ground truth provided.

In this evaluation, we use two pre-trained word embedding models and one task-specific model in order to decide the best configuration to incorporate in our pipeline:

- Word2vec:[16] embedding vectors pre-trained on part of Google News dataset, comprising about 100 billion words. The model contains 300-dimensional vectors for 3 million words and phrases [13], although in the experiments presented here only words are considered.
- fastText:[17] embedding vectors pre-trained on Wikipedia 2017, UMBC webbase corpus and statmt.org news dataset, comprising about 16 billion words [23].

[16] https://code.google.com/archive/p/word2vec/, accessed October 2020.
[17] https://github.com/facebookresearch/fastText, accessed October 2020.

As in the previous case, the vectors are composed of 300 dimensions.

- WikiTables: task-specific model using skip-gram Word2vec [24] trained on the Wikipedia Tables corpus, containing 1.6 million Wikipedia relational tables [25]. The corpus was pre-processed as mentioned in Section III-A1, splitting CamelCase and hyphenated words, removing punctuation, and converting text to lowercase. For every table in this corpus, all the names of the columns were extracted and treated as an input document to train Word2vec.[18] A second model was created for the content of the cells. In this case, all the attribute values in a column were considered as an input document to train the model. Thus, we have two separate word embedding models to calculate the similarity between names of the columns and the content of the cells. Again, vectors are composed of 300 dimensions.

In the case of fastText, we also conducted experiments with a pre-trained model containing subword information, but the results obtained were worse than the model presented above. We decided to remove these results from the following section for the sake of simplicity.

We computed the coverage of each of these models, calculated as the percentage of tokens in the 1,000 tables bechmark dataset that has a representation in the model. In the case of Google Word2vec, the coverage for names of the columns and content of the cells is 83.02% and 78.18% respectively. fastText provides a coverage of 87.69% and 80.91% for columns and cells. Finally, the coverage of WikiTables is 71.54% for columns and 78.19% for cells. These values reflect that fastText is the model offering the largest coverage, whereas WikiTables provides the lowest results. As mentioned in Section III-A1, the Levenshtein distance is used as a backup strategy when a word is not represented in the model.

2) RESULTS AND DISCUSSION

This section reports the precision of top-k table searches at different k. More precisely, P@10 and P@50 where computed, corresponding to the number of relevant results among the top 10 and top 50 documents retrieved respectively. This measure is in accordance with web-scale information retrieval systems, where thousands of relevant documents are available but no user will be interested in reading all of them. The final score is computed as the average precision for the 1,000 queries carried out, one for each table.

The experiments include the three word embeddings models mentioned above. In addition, we tested a baseline using BM25 [26], a classical ranking function widely employed by search engines to estimate the relevance of documents to a given search query. The implementation of the baseline was carried out using Apache Solr.[19]

In order to identify the best value for α, we tested each embedding model with different values for this parameter, from 0 to 1 inclusive, in 0.1 increments. In the case of the baseline, three different queries were employed in Apache Solr to simulate the ranking experiment done with the embedding vectors: a query that uses only the content of the cells (equivalent to $\alpha = 0.0$), a query that uses only the names of the columns (equivalent to $\alpha = 1.0$) and, finally, a query containing both (equivalent to $\alpha = 0.5$). Fig. 12 and Fig. 13 show the results for Google pre-trained Word2vec (*gl*), pre-trained fastText (*ft*), task-specific Word2vec trained on Wikipedia Tables (*wt*), and BM25 (*bm*).

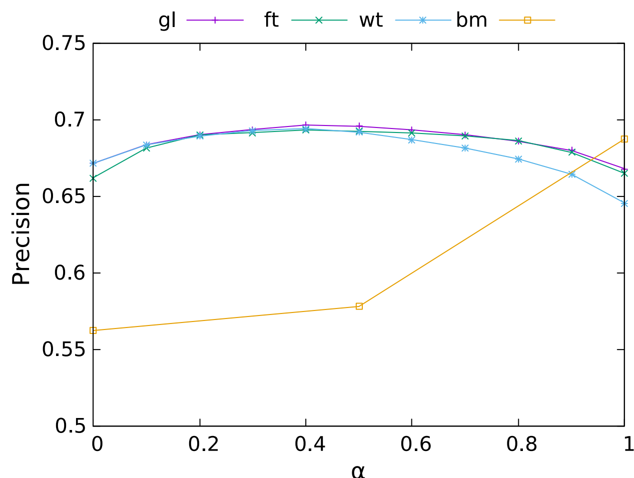

FIGURE 12. P@50 for each of the embedding models (*gl*, *ft*, and *wt*) and the baseline (*bm*).

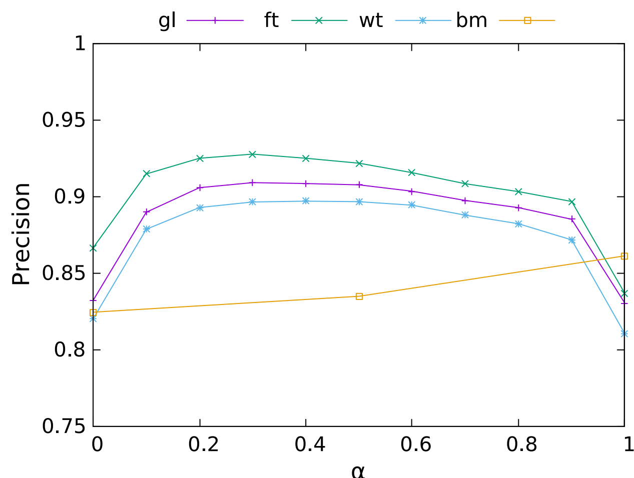

FIGURE 13. P@10 for each of the embedding models (*gl*, *ft*, and *wt*) and the baseline (*bm*).

The results at P@50 show that the three word embeddings models perform very close for every α value. The best result is obtained by Google pre-trained embeddings with P@50=0.6964 and $\alpha = 0.4$, which means that the content of the cells have a higher relevance in the final similarity score. The other two embedding models reach also its best performance with $\alpha = 0.4$. In the case of the BM25 baseline, it is worth noting that the results using only the content

[18] https://radimrehurek.com/gensim/models/word2vec.html

[19] https://lucene.apache.org/solr/.

of the cells are significantly lower than the word embeddings approaches. These results slightly improve when both names of columns and content of cells are taken into account ($\alpha = 0.5$), but it achieves the best results when only the name of the columns are considered ($\alpha = 1.0$), outperforming the other models.

Results at P@10 are encouraging in terms of the performance of the word embeddings models. The best results (0.9276) are obtained by the fastText model with $\alpha = 0.3$. This result improves 7.71% the performance of the baseline. The comparison between the three embedding models shows that its performance correlates with the coverage previously described. This highlights the importance of having a large enough corpus to properly build an effective word embeddings model for this task.

For the end user, the most important aspect is to improve the precision of the first results obtained by our approach. Therefore, P@10 is the measure that we should try to maximise. For this reason, we choose the best performing configuration in this evaluation (fastText model with $\alpha = 0.3$) to tune the system for the API generation experiments described in the following section.

The retrieval phase described here was omitted in the running example, since we started directly having the tables we wanted to integrate. Nevertheless, we can compute the similarity $sim(t_1, t_2)$ between these tables to see how they would rank in an hypothetical retrieval task such as the one presented in this section. The similarity between the two tables from Chicago used in the join operation is 0.2872. This value reveals that the tables present an average low similarity, which is an expected result for join operations since most of their columns are not related. It should be noted that the formula is computing the similarity between all possible pairs of columns, which in this case involves 33 and 39 columns for a total of 1,287 pairs. Following our approach the tables were considered as joinable since one pair of columns achieved a similarity value over 0.95 as described in Section III-A2. The similarity between the resulting joined table (71 columns) and the New York table (4 columns) is 0.6304, showing significantly higher similarity than the previous case. The reason is that the number of column pairs to compare is lower (284) and there are more related pairs of columns. Indeed, all the columns in the New York city table presented a similarity over 0.75 with the corresponding columns in the Chicago table, resulting in the union of both tables as established in our methodology.

Although the task proposed in this section is similar to that defined by Nargesian *et al.* [22], the results are not directly comparable. While they try to maximise the performance on their own definition of "unionability" (involving all the unionable columns of a table), our goal is to address both union and join operations where tables have at least one matching column in common. Taking into account these differences, the authors obtained a P@10 value of around 0.8 with an approach based solely in word embeddings, which improved to about 0.95 when combined with semantic information from YAGO [27] and the actual set of values from the tables. Thus, our best result (0.9276) is in a similar range to those obtained by Nargesian *et al.* with their best performing ensemble of features, whereas we are using only embedding information at this stage.

B. GENERATING WEB APIs FOR ACCESSING INTEGRATED OPEN DATA

In this section, we provide an evaluation of the approach for automatically generating Web APIs to access integrated data. This evaluation measures the quality by considering the number of correctly generated APIs for a set of input tabular datasets.

1) EXPERIMENTAL SETUP

We use again the benchmark described by Nargesian *et al.* for this evaluation. As mentioned before, this dataset was generated starting from a set of 32 base tables on which the authors performed different selections and projections on various sizes to obtain the final set of 5,000 tables. From this set of 32 base tables, only 27 were involved in the generation of the 1,000 random subset used in the previous evaluation (see Section IV-A). With the aim of aligning both experiments, we focused on these 27 base tables for the evaluation done in this section.

In order to test the performance of our approach on both union and join operations, each table was divided into three subtables by using projection and filtering as reverse operations of join and union, respectively. The goal of the system in this experiment is to automatically generate a Web API for accessing the integration of these subtables by using first join and then union operations.

Two experiments were conducted either considering columns names or not when calculating the similarity between tables. The rationale behind this is to avoid bias regarding the fact that subtables coming from the same table have the same column names, and this would not be the usual situation in a real scenario. Thus, the first experiment considers only the content of the cells, and the second experiment uses both content and column names for computing the similarity between tables.

2) RESULTS AND DISCUSSION

The results of the experiments for evaluating the generated Web APIs are shown in Table 6. The column *Original Web API functions* represents the expected functions to be generated for each of the 27 base tables in the benchmark. The columns *Correctly generated Web API functions* indicates the number of functions that were correctly generated by our proposal in the two settings proposed: with and without column names. *Precision* is defined in this case as the number of functions of the API generated from the integrated tables compared to the number of functions offered by the API from the original tables. In the final results, we consider both micro-averaged precision (i.e. aggregating the functions of all the tables to compute the average) and macro-averaged

TABLE 6. Results of the Web API generation experiments.

Table	Original Web API functions	Correctly generated Web API functions (without column names)	Precision	Correctly generated Web API functions (with column names)	Precision
1	12	10	0.83	11	0.92
2	7	6	0.86	7	1.00
3	7	3	0.43	7	1.00
4	13	3	0.23	9	0.69
5	6	6	1.00	6	1.00
6	22	21	0.96	22	1.00
7	6	6	1.00	6	1.00
8	13	13	1.00	13	1.00
9	13	4	0.31	11	0.85
10	14	14	1.00	14	1.00
11	19	16	0.84	17	0.90
12	23	23	1.00	14	0.61
13	20	20	1.00	19	0.95
14	21	21	1.00	17	0.81
15	23	13	0.57	23	1.00
16	13	8	0.62	8	0.62
17	16	3	0.19	3	0.19
18	9	9	1.00	9	1.00
19	28	24	0.86	25	0.89
20	13	1	0.08	4	0.31
21	19	8	0.42	19	1.00
22	26	24	0.92	26	1.00
23	11	6	0.55	9	0.82
24	19	16	0.84	16	0.84
25	17	13	0.77	17	1.00
26	22	22	1.00	22	1.00
27	12	10	0.83	12	1.00
All	424	323		366	
	Micro-averaged precision		0.76		0.86
	Macro-averaged precision		0.74		0.87

precision (i.e. precision is computed for every table and then averaged). Macro-averaged precision allows treating all the APIs generated equally regardless of the number of functions they have.

The value of micro-averaged precision obtained is 0.76 when column names are not consider. This value rises to 0.86 when the column names are taken into account in the similarity measure formula. Macro-averaged precision is 0.74 if column names are not considered and 0.87 otherwise. In both cases, the use of column names improves the results as expected, since they are clear cues to identify whether two columns match in the similarity algorithm proposed. It is worth noting that using only the content of the cells allows generating the correct API functions about 75% of the time.

In order to clarify why the system fails to generate the correct API functions in some situations, we analysed in more detail the specific case of table 3, where our approach obtained perfect precision using the names of the columns but dropped to 0.43 when using only the content of the cells. This table contains data about ridership from Chicago Transit Authority.[20] The API generated for the original dataset has to include the 7 functions that are shown in Fig. 14.

FIGURE 14. Functions of the Web API generated to access data from table 3.

The API automatically generated considering column names has the same 7 functions, whereas the API generated using only content of the cells fails to produce the following 4 functions:

```
/alightings/{alightings}
/boardings/{boardings}
/cross_street/{cross_street}
/on_street/{on_street}
```

When only the content of the table is considered, the similarity algorithm incorrectly matches the columns "alightings" and "boardings" with a value of 0.9996, while the similarity with the correct column "alightings" is 0.9991. The reason is that both columns have very similar values as the concepts are closely related ("boarding" means going into a bus, while "alighting" means going out of the bus), making it difficult for the algorithm to discriminate between the two cases.

[20]https://www.transitchicago.com/

TABLE 7. Description of the dataset and execution time performance (in seconds) for the integration and generation processes.

Table name	Rows	Columns	Rows selected	Integration time	Generation time
Salmonella tests	13	3	13 (100%)	0.0028	9.7
Population	441	5	30 (6.8027%)	0.0059	9.7
Travel data	1,853	20	70 (3.7776%)	0.0608	10.3
Biodiversity	20,017	12	30 (0.1499%)	0.0184	10.3
Street lights	27,409	15	20 (0.0730%)	0.0138	10.5
International payments	245,107	17	20 (0.0082%)	0.0133	10.9
Traffic state	3,565,683	4	20 (0.0006%)	0.0030	13.4
Voter data	7,517,745	46	700 (0.0093%)	0.6762	29.2
Wholesale trade sales	8,752,568	17	450 (0.0051%)	0.3176	35.1
Employee earnings	21,072,480	18	150 (0.0007%)	0.1318	80.0

Another example is the column ''cross street'', which is incorrectly matched to ''on street'' with a similarity of 0.9914, while the similarity with the correct column ''cross street'' is 0.9833. Again, the reason is that both columns have similar values as the concepts are related (''on street'' refers to the street where the bus stop is, while ''cross street'' refers to the nearest crossing street).

In this situation, when the contents of the cells are very similar, it is required the use of additional metadata such as the names of the columns to achieve a good performance in the automatic generation of the API.

In the case of the running example, using the best configuration (fastText model with $\alpha = 0.3$) all the API functions expected were correctly generated for the resulting tables after join (72 functions, including the *publisher* column) and union (5 functions, including also the *publisher* column) operations, achieving a perfect precision of 1.00 in this experiment.

C. EXECUTION TIME PERFORMANCE

This section analyses the execution time performance of the system proposed, providing a reference on the time required to carry out the integration and generation processes from a CSV file, and discussing their implications in a production environment.

1) EXPERIMENTAL SETUP

In order to evaluate the execution time performance of the integration and API generation processes, two experiments were carried out using 10 different CSV datasets.[21] These files were retrieved from different open data platforms, representing an heterogeneous set in terms of their size and nature. The content of the tables includes texts, numbers, and dates. The amount of rows in the tables ranges from 13 (table ''Salmonella tests'') to over 21 millions (table ''Emloyee earnings''), and from 3 columns (table ''Salmonella tests'') to 20 (tables 'Travel data'' and ''Voter data''), resulting in a minimum of 39 cells and a maximum of 379,304,640. The specific number of rows and columns for each CSV file is shown in Table 7.

[21] Datasets available in GitHub repository: https://github.com/cgmora12/DataIntegration2API#example-datasets.

The measurement of the execution time of the experiments was carried out in a desktop computer running Windows 10, equipped with an Intel i7 processor and 16 GB of RAM.

2) RESULTS AND DISCUSSION

First we evaluated the time elapsed during the integration process. In this phase calculating the embeddings for each column of a table is the bottleneck of the task. Once the embeddings are obtained, computing the similarity between two tables is reduced to calculate the inner product between 300-dimensional vectors representing their columns, which can be computed in a minimal time. Thus, in this first experiment we have focused on determining the time taken to calculate the embedding vectors for each table.

This task is time consuming when the tables have a large number or rows, since the embedding of each column is computed as the average of the embeddings of its content. For instance, in the example table ''Traffic state'' containing 3,565,683 rows and 4 columns, the process of obtaining the embedding vectors takes 423.41 seconds in the hardware setting mentioned above, which can be unsuitable for a production environment.

To reduce the time required for this process, we first analysed to what extent the number of rows could be reduced without affecting the embedding representation of the table, taking into account that in many cases columns contain repeated values that could be removed seamlessly. For each table we randomly choose samples of rows of different sizes[22] (ranging from 1 to 20,000 in increments of 10) and obtained the corresponding embeddings. Then, the similarity measure $sim(t_1, t_2)$ defined in Section IV-A1 was used to compare how similar were the reduced version of the table and the full version containing all the rows. The results of this experiment revealed that, in all the cases, we were able to obtain a reduced version of the table that was 99% similar to the original one by considering a small sample of the rows.

Table 7 shows the number of rows and the percentage it represents from the original (column *Rows selected*) that was required for each table to exceed the 99% similarity threshold. For instance, table ''Traffic state'' contains over 3 million rows, but the reduced version has only 20 rows as

[22] Except for ''Salmonella tests'', which contained only 13 rows.

a result of having a small number of columns (only 4) and a large number of repeated values in them. The table that required a larger number of samples was "Voter data", with 700 rows. In all the cases, the time taken to obtain the embedding representation of the reduced table was below one second (column *Integration time*). For the largest tables (above 200,000 rows), the percentage of rows kept from the original was below 0.01%. This value could be established as a fixed threshold in a production environment to ensure that the tables can be processed in real time while retaining (at 99%) their original representation in the word embedding space.

Regarding the evaluation of the API generation process, the results presented in Table 7 (column *Generation time*) show that the automatic process takes between 9.7 and 80.0 seconds to generate a complete Web API, including the related models and interactive OpenAPI documentation. This evaluation was carried out using all the rows in the tables to identify how this number affects the execution time of the generation module. If we took the reduced tables of the previous experiment, where the largest table contains 700 rows, the generation time would be comprised between 9.7 and 10.3 seconds in all cases.

Fig. 15 shows a linear behaviour considering the number of rows of the tables. The process is able to generate the Web API and its related components (models and documentation) in an average time of 16 seconds. The results indicate that, as the number of rows in the source file increases, the time to generate the Web API also increases in a ratio of 3 seconds per million rows.

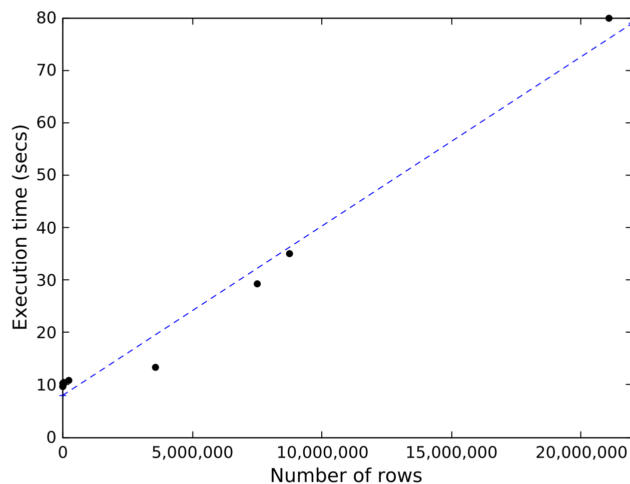

FIGURE 15. Generation time of the API depending on the size of the tables.

Regarding the performance of the generated APIs in returning the results, when the number of rows is really high (such in the case of "Employee earnings") the API takes a time similar to the generation performance shown in Table 7. This situation can be overcome by the pagination of the results using "limit" and "offset" parameters: "limit" indicates the maximum results to show, whereas "offset" indicates the starting point of results bypassing the records until the specified offset. This pagination auto-generated by our APIfication approach optimises the performance of the API and reduces the time to get the results. With this mechanism, APIs are able to return results from datasets of up to 21 million rows in less than 4 seconds. For example, a query that specifies a limit of 10,000 results to the Web API that manages the largest dataset ("Employee earnings") takes around 3 seconds to return the results to the browser.

V. RELATED WORK

The following paragraphs summarise the existing related work. Specifically, we focus on two areas: (i) word embeddings for data integration; and (ii) Web API generation for open data access.

A. WORD EMBEDDINGS AND TABULAR DATA INTEGRATION

In recent years, word embeddings have enjoyed widespread use in a variety of semantic tasks in the field of Natural Language Processing, such as sentiment analysis [28], machine translation [29], text classification [30], and dialog systems [31].

Prior works related to tabular data considered word embeddings as a means to represent the content of the tables. In the task of table retrieval, consisting on answering a search query with a ranked list of tables, Zhang *et al.* [32] used pre-trained word and entity embeddings combined with different similarity measures to beat a strong learning to rank baseline. The work by Deng *et al.* [20] considered different table elements (caption, column headings, and cells) to train word embeddings that were utilised in three table-related tasks: row population, column population, and table retrieval. In a similar vein, Nargesian *et al.* [22] defined a semantic measure based on word embeddings that were trained on Wikipedia documents. The authors defined natural language domains and statistical tests between the vectors that were used to evaluate the likelihood that two attributes were from the same domain.

In the work presented in this paper, we follow a similar approach to previous research using pre-trained word embeddings, but introducing as a novelty the inclusion of task-specific embeddings based on a large dataset of tabular data. More importantly, unlike previous works that treat data integration as an isolated process, we include this procedure in a pipeline to automatically generate Web APIs for the integrated data.

B. WEB APIs FOR OPEN DATA ACCESS

Web APIs are created to make accessing open data easier for developers. However, this process has been usually done manually [33], [34] as a time-consuming task.

The automatic generation of APIs has been addressed in recent studies. EMF-REST [35] is a framework for generating Web APIs that needs a model of the API to create it, requiring users to build the model by themselves since it is not generally

available. Another work [36] focused on guiding developers in creating Web APIs for accessing required data. However, these approaches are not targeting the access and reuse of integrated open data coming from different sources.

Other works propose the use of model-driven approaches for API definitions. The models are used to represent the Web API definition, offering a better visualisation of the API operations [37], [38]. Also, the metamodel of the API definition is used to simplify the transformation between the API and its definition to include it in the OpenAPI initiative [39]. The API2MoL engine [40] creates bridges between APIs and model-driven engineering, with the objective of creating models from the APIs for facilitating the management of a plethora of APIs.

These model-based approaches for generating Web APIs can be used to represent and generate the Web API documentation, but they are not employed to generate the whole Web API to access integrated data as proposed in this paper.

VI. CONCLUSION AND FUTURE WORK

In this paper we have presented an approach to address the problem of accessing integrated open data from different sources by using a unique end point. Specifically, we have proposed an APIfication approach which aims to facilitate the integration and access to tabular datasets. Our APIfication approach has two parts: (i) a word embeddings-based approach that uses column similarity to determine which datasets can be integrated by using union and join operators; and (ii) a model-driven approach for automatically generating a Web API to access and reuse open data in an integrated manner, as well as the documentation of the Web API to support users in consuming the integrated data easily through a Web interface.

In the experiments conducted, the proposed similarity measure based on word embeddings achieved precision values over 0.92 in the task of retrieving unionable/joinable tables given a query table. When this measure was included in the API generation pipeline, our approach was able to automatically integrate the tables and generate the expected functions with a micro-averaged precision of 0.76 using only the content of the cells, and 0.86 precision when the names of the columns were also taken into account. These results are promising, even more so considering that the whole data integration and API generation process is fully automated. The execution time performance experiments revealed that selecting a small number of rows allowed to obtain the word embeddings representation for large tables in real time, while keeping the table representation almost unchanged in the vector space (99% similarity with respect to the original table). Additionally, the API generation procedure for tables of more than one million rows took around 10 seconds to complete, which makes the whole pipeline suitable for a production scenario.

As a future work, we plan to extend our approach by considering more operators for data integration in addition to union and join. We also plan to develop mechanisms to guide users in the application of the approach, for example, by defining GUIs to allow developers to provide semantic hints about the open data to be integrated. Moreover, we plan to investigate how metadata coming from the W3C CSV on the Web Working Group[23] can improve our approach, as well as whether it is a motivation for publishers to properly describe their tabular open data.

Regarding the similarity module, future experiments are required in order to determine the best performing combination of models, i.e. using one model (e.g. Word2vec) to calculate the similarity between column names and another one (e.g. fastText) for the content values. The backup strategy based on Levenshtein distance can also be improved by using subword embeddings that can handle the problem of out-of-vocabulary terms, such as the subword version of fastText mentioned before, or models using Byte-Pair Encoding (BPE) [41]. This approach could benefit the system when the content of the tables include numbers and dates, as it provides better coverage than word-based models that could offer limited representation of numeric values. An interesting path for further research is the use of contextual word embedding models such as BERT [42] and its derivatives (e.g. ALBERT [43], RoBERTa [44], and DistilBERT [45]). These models provide different vector representations for a term in the embedding space depending on the context (surrounding terms) where it occurs. They have demonstrated to achieve state-of-the-art results on a number of language understanding tasks, including question answering and natural language inference. Another issue that deserves attention is the improvement of the WikiTables task-specific model by gathering additional tabular examples. In the experimental evaluation we showed that this model offered a lower coverage than the other pre-trained models, which could be affecting its performance. Finally, another line of future work is the combination of the similarity functions provided by the word embeddings models with traditional information retrieval ranking functions such as BM25, which demonstrated to obtain better results than the word embedding models in those experiments where only the content of the tables was taken into account.

REFERENCES

[1] M. S. Altayar, "Motivations for open data adoption: An institutional theory perspective," *Government Inf. Quart.*, vol. 35, no. 4, pp. 633–643, Oct. 2018.

[2] H. Dong, G. Singh, A. Attri, and A. El Saddik, "Open data-set of seven canadian cities," *IEEE Access*, vol. 5, pp. 529–543, 2017.

[3] C. Bizer, T. Heath, and T. Berners-Lee, "Linked data: The story so far," in *Linked Data: The Story so Far, in: Semantic Services, Interoperability and Web Applications: Emerging Concepts*. Hershey, PA, USA: IGI Global, 2011, pp. 205–227.

[4] S. Neumaier, J. Umbrich, and A. Polleres, "Automated quality assessment of metadata across open data portals," *J. Data Inf. Qual.*, vol. 8, no. 1, pp. 1–29, Nov. 2016, doi: 10.1145/2964909.

[5] E. Muñoz, A. Hogan, and A. Mileo, "Using linked data to mine RDF from wikipedia's tables," in *Proc. 7th ACM Int. Conf. Web Search Data Mining - WSDM*, 2014, pp. 533–542, doi: 10.1145/2556195.2556266.

[23] https://www.w3.org/TR/tabular-data-model/

[6] *Open Data Maturity Report 2019*, Publications Office of the European Union, European Commission, Brussels, Belgium, 2019.
[7] *Open Government Data Report: Enhancing Policy Maturity for Sustainable Impact, OECD Digital Government Studies (2018)*, Organisation for Economic Co-operation and Development, OECD, Paris, France, 2018.
[8] M. Arenas, F. Maturana, C. Riveros, and D. Vrgoč, "A framework for annotating CSV-like data," *Proc. VLDB Endowment*, vol. 9, no. 11, pp. 876–887, Jul. 2016.
[9] Y. Doi and M. Toyama, "ToT for CSV: Accessing open data CSV files through SQL," in *Proc. 21st Int. Conf. Inf. Integr. Web-based Appl. Services*, Dec. 2019, pp. 423–429.
[10] R. J. Miller, "Open data integration," *Proc. VLDB Endowment*, vol. 11, no. 12, pp. 2130–2139, 2018.
[11] A. Halevy, A. Rajaraman, and J. Ordille, "Data integration: The teenage years," in *Proc. 32nd Int. Conf. Very large data bases*, 2006, pp. 9–16.
[12] D. Abadi *et al.*, "The seattle report on database research," *ACM SIGMOD Rec.*, vol. 48, no. 4, pp. 44–53, Feb. 2020, doi: 10.1145/3385658.3385668.
[13] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," in *Proc. 26th Int. Conf. Neural Inf. Process. Syst.*, Lake Tahoe, NV, USA, vol. 2, 2013, pp. 3111–3119.
[14] K. Braunschweig, J. Eberius, M. Thiele, and W. Lehner, "The state of open data limits of current open data platforms," in *Proc. 21st. WWW Conf.*, 2012, pp. 1–6.
[15] Z. S. Harris, "Distributional structure," *Word*, vol. 10, nos. 2–3, pp. 146–162, Aug. 1954.
[16] S. Gupta, T. Kanchinadam, D. Conathan, and G. Fung, "Task-optimized word embeddings for text classification representations," *Frontiers Appl. Math. Statist.*, vol. 5, p. 67, Jan. 2020.
[17] V. I. Levenshtein, "Binary codes capable of correcting deletions, insertions and reversals," *Sov. Phys. Doklady*, vol. 10, no. 8, pp. 707–710, 1966.
[18] J. S. Cuadrado, E. Guerra, and J. de Lara, "AnATLyzer: An advanced IDE for ATL model transformations," in *Proc. 40th Int. Conf. Softw. Eng., Companion Proceeedings*, May 2018, pp. 85–88, doi: 10.1145/3183440.3183479.
[19] A. Srai, F. Guerouate, N. Berbiche, and H. Drissi, "An MDA approach for the development of data warehouses from relational databases using ATL transformation language," *Int. J. Appl. Eng. Res.*, vol. 12, pp. 3532–3538, 2017.
[20] L. Zhang, S. Zhang, and K. Balog, "Table2 Vec: Neural word and entity embeddings for table population and retrieval," in *Proc. 42nd Int. ACM SIGIR Conf. Res. Develop. Inf. Retr.*, Jul. 2019, pp. 1029–1032.
[21] A. Das Sarma, L. Fang, N. Gupta, A. Halevy, H. Lee, F. Wu, R. Xin, and C. Yu, "Finding related tables," in *Proc. Int. Conf. Manage. Data - SIGMOD*, 2012, pp. 817–828.
[22] F. Nargesian, E. Zhu, K. Q. Pu, and R. J. Miller, "Table union search on open data," *Proc. VLDB Endowment*, vol. 11, no. 7, pp. 813–825, Mar. 2018.
[23] P. Bojanowski, E. Grave, A. Joulin, and T. Mikolov, "Enriching word vectors with subword information," *Trans. Assoc. for Comput. Linguistics*, vol. 5, pp. 135–146, Dec. 2017.
[24] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," in *Proc. 1st Int. Conf. Learn. Represent.*, 2013, pp. 1–12.
[25] C. S. Bhagavatula, T. Noraset, and D. Downey, "Tabel: Entity linking in Web tables," in *The Semantic Web—ISWC*. Cham, Switzerland: Springer, 2015, pp. 425–441.
[26] S. E. Robertson, S. Walker, S. Jones, M. M. Hancock-Beaulieu, and M. Gatford, *Okapi at TREC-3*, vol. 109. Gaithersburg, MD, USA: NIST Special Publication Sp, 1995.
[27] F. M. Suchanek, G. Kasneci, and G. Weikum, "Yago: A core of semantic knowledge," in *Proc. 16th Int. Conf. World Wide Web - WWW*, 2007, pp. 697–706.
[28] M. Giatsoglou, M. G. Vozalis, K. Diamantaras, A. Vakali, G. Sarigiannidis, and K. C. Chatzisavvas, "Sentiment analysis leveraging emotions and word embeddings," *Expert Syst. Appl.*, vol. 69, pp. 214–224, Mar. 2017.
[29] W. Y. Zou, R. Socher, D. Cer, and C. D. Manning, "Bilingual word embeddings for phrase-based machine translation," in *Proc. Conf. Empirical Methods Natural Lang. Process.*, 2013, pp. 1393–1398.
[30] M. J. Kusner, Y. Sun, N. I. Kolkin, and K. Q. Weinberger, "From word embeddings to document distances," in *Proc. 32nd Int. Conf. Mach. Learn.*, 2015, pp. 957–966.
[31] G. Forgues, J. Pineau, J.-M. Larchevêque, and R. Tremblay, "Bootstrapping dialog systems with word embeddings," in *Proc. NIPS Workshop Modern Mach. Learn. Natural Lang. Process.*, vol. 2, 2014, pp. 1–5.
[32] S. Zhang and K. Balog, "Ad hoc table retrieval using semantic similarity," in *Proc. World Wide Web Conf. World Wide Web - WWW*, 2018, pp. 1553–1562.
[33] I. Hopkinson, S. Maude, and M. Rospocher, "A simple API to the knowledgestore," in *Proc. Int. Conf. Developers*, vol. 1268, 2014, pp. 7–12. [Online]. Available: http://dl.acm.org/citation.cfm?id=2878379.2878381
[34] M. Rittenbruch, M. Foth, R. Robinson, and D. Filonik, "Program your city: Designing an urban integrated open data API," in *Proc. Cumulus Conf., Open Helsinki–Embedding Design Life*, 2012, pp. 24–28.
[35] H. Ed-douibi, J. L. C. Izquierdo, A. Gómez, M. Tisi, and J. Cabot, "EMF-REST: Generation of RESTful APIs from models," in *Proc. 31st Annu. ACM Symp. Appl. Comput. SAC*, 2016, pp. 1446–1453.
[36] R. Queirós, "Kaang: A RESTful API Generator for the Modern Web," in *Proc. 7th Symp. Lang., Appl. Technol.*, vol. 62, 2018, pp. 1:1–1:15.
[37] H. Ed-douibi, J. L. C. Izquierdo, F. Bordeleau, and J. Cabot, "WAPIml: Towards a modeling infrastructure for Web APIs," in *Proc. ACM/IEEE 22nd Int. Conf. Model Driven Eng. Lang. Syst. Companion (MODELS-C)*, Sep. 2019, pp. 748–752.
[38] H. Ed-douibi, J. L. Cánovas, and J. Cabot, "OpenAPItoUML: A tool to generate UML models from OpenAPI definitions," in *Web Engineering*. Cham, Switzerland: Springer, 2018, pp. 487–491.
[39] H. Ed-douibi, J. L. Cánovas, and J. Cabot, "Example-driven Web API specification discovery," in *Modelling Foundations and Applications*. Cham, Switzerland: Springer, 2017, pp. 267–284.
[40] J. L. Cánovas Izquierdo, F. Jouault, J. Cabot, and J. García Molina, "API2MoL: Automating the building of bridges between APIs and model-driven engineering," *Inf. Softw. Technol.*, vol. 54, no. 3, pp. 257–273, Mar. 2012. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0950584911001984
[41] B. Heinzerling and M. Strube, "Bpemb: Tokenization-free pre-trained subword embeddings in 275 languages," in *Proc. 11th Int. Conf. Lang. Resour. Eval. (LREC)*, Miyazaki, Japan, 2018, pp. 2989–2993.
[42] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," in *Proc. 2019 Conf. North Amer. Chapter Assoc. Comput. Linguistics*. Minneapolis, Minnesota: Association for Computational Linguistics, 2019, pp. 4171–4186.
[43] Z. Lan, M. Chen, S. Goodman, K. Gimpel, P. Sharma, and R. Soricut, "ALBERT: A lite BERT for self-supervised learning of language representations," *CoRR*, vol. abs/1909.11942, 2019. [Online]. Available: http://arxiv.org/abs/1909.11942
[44] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov, "Roberta: A robustly optimized BERT pretraining approach," 2019, *arXiv:1907.11692*. [Online]. Available: https://arxiv.org/abs/1907.11692
[45] V. Sanh, L. Debut, J. Chaumond, and T. Wolf, "DistilBERT, a distilled version of BERT: Smaller, faster, cheaper and lighter," *CoRR*, vol. abs/1910.01108, 2019. [Online]. Available: http://arxiv.org/abs/1910.01108

CÉSAR GONZÁLEZ-MORA is currently pursuing the Ph.D. degree with the Web and Knowledge Research Group, Department of Software, University of Alicante, Spain. His research interests include open data, Web augmentation, and the semantic Web and application programming interfaces. His work is funded by a contract with the Generalitat Valenciana of Spain and the European Social Fund for predoctoral training.

DAVID TOMÁS is currently a Lecturer with the Department of Software and Computing Systems, University of Alicante, Spain. He is the author of more than 70 scientific publications in international conferences and journals. He has participated in over 20 public and private projects, including EU-funded QALL-ME, FIRST, and SAM projects. His research interests include information retrieval, knowledge representation, information extraction, question answering, sentiment analysis, recommender systems, log analysis, and machine learning approaches to text categorisation.

IRENE GARRIGÓS is currently an Associate Professor with the Department of Software and Computing Systems, University of Alicante, Spain. She is also the Head of the Web and Knowledge Research Group, University of Alicante. Her research interests include open data, Web augmentation, Web modeling languages, personalization, and application programming interfaces.

JOSÉ JACOBO ZUBCOFF presents a wide teaching and research experience in the field of statistics, data mining, and its application to biology. He has more than 100 publications in which he has dealt with obtaining knowledge from a data source. He has done research in various fields of science, both in computing, biology, medicine, education, and social sciences. In addition, he has directed and participated in more than 20 competitive public projects financed by the Ministry of Economy and Competitiveness, the Generalitat Valenciana, the University of Alicante, and European and private projects, all of them contributing his knowledge about data analysis, data mining, and aiming at the democratization of knowledge.

JOSE-NORBERTO MAZÓN is currently an Associate Professor with the Department of Software and Computing Systems, University of Alicante, Spain. He is the author of more than 100 scientific publications in international conferences and journals. His research interests include open data, business intelligence in big data scenario, design of data-intensive Web applications, smart cities, and smart tourism destinations. He is also the Chair of the Torrevieja's Venue of the University of Alicante.