sHINER Manual

TU Eindhoven[1], SmartDataLake project[2]

[1] Database Research Group,
Department of Mathematics and Computer Science,
Eindhoven University of Technology,
Eindhoven, Netherlands

[2] European Union's Horizon 2020
Research and innovation programme
Grant Agreement No 825041
https://smartdatalake.eu/

Abstract. This document describes the sHINER component of the SmartDataLake project. The sHINER component is responsible for the Entity Resolution task of the project. In this component, we use Graph Generating Dependencies to encode entity resolution rules. In this manual we concentrate on the technical details, i.e., how to compile the code and use the component.

1 Introduction

1.1 Graph Generating Dependencies using G-CORE interpreter on Spark

The Graph Generating Dependencies is a new class of graph dependencies proposed for property graphs inspired by the tuple- and equality-generating dependencies for relational data. More information on the Graph Generating Dependencies and its syntax refer to the paper at [3].

This component is able to run the validation of the input GGDs and "fix" it by generating new nodes/edges in the graph. The project uses the G-Core interpreter on Spark for querying the defined graph patterns. The G-Core project is implemented and maintained by the LDBC council in their repository [3]. For more information on G-Core query language refer to the original project page or check their publication at [1]. Some operators on the G-Core interpreter used for the GGDs were added: SELECT, UNION and OPTIONAL queries. To add these operators we changed the Spoofax language file as well as on the compilation class file of these new types of queries.

In order to validate the differential constraints in the GDDs we added a similarity join operator for Jaccard and Edit Similarity by using the methods from Dima [4] and the Vernica Join method (code provided by the Athena Research

[3] https://ldbc.github.io/gcore-spark/
[4] https://github.com/TsinghuaDatabaseGroup/Dima

Center). These methods were added using SparkSQL Extensions API. The Figure 1 shows the general architecture of the sHINER component and how it was implemented.

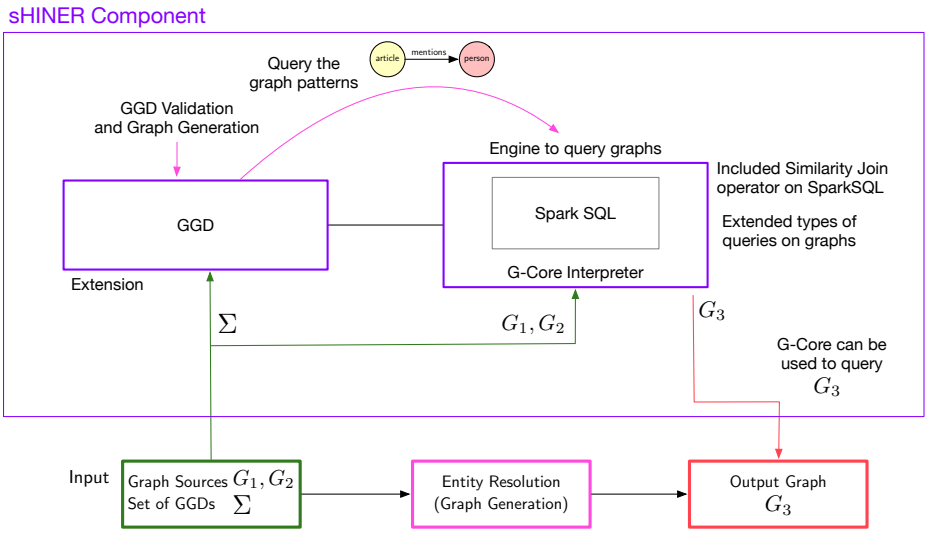

Fig. 1: Architecture of sHINER

This project is still under development and this manual will be updated as the project is modified.

2 Getting Started

In this section we describe how the project can be built.

3 Prerequisites

The sHINER component was developed on 64-bit Linux. The prerequisites to build this project on Linux are:

1. A 64-bit Linux distribution
2. Java JDK version 8
3. Apache Maven (for building the project)
4. Apache Spark 2.4.6
5. Guice library jar file[5]

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

Further library dependencies such as Scala and JSON libraries are integrated by using Apache Maven. Specifications about these libraries are in the pom.xml of each project.

To build the project we need to build both the sHINER[6] component and a Spark 2.4.6 installation. To install Spark follow the installation steps at Spark official website: `https://spark.apache.org/`.

To build the sHINER use the following instructions:

```
git clone https://github.com/smartdatalake/gcore-spark-ggd.git

cd gcore-spark-ggd

mvn install
```

Once the project is built we can run sHINER component in 3 possible ways: (1) Using command line extending GCore; (2) Command line using a configuration file (recommended for Entity Resolution evaluation), and (3) REST API. We are going to describe how we can possibly run each one of these options.

3.1 1 - Command-line extending G-Core API

In this mode of running the sHINER component it extends the command-line application of the G-Core language interpreter and runs the Validation algorithm for the setted GGDs. This mode is recommended for those who want to give a quick try on the component and its features, or check if the component is running correctly in the installed environment. In this mode all the graph pattern queries required by the GGDs validation algorithm will run using SparkSQL. Start the command-line application using the following commands.

```
./bin/spark-submit \
--class ggd.ERRunner \
--master local[2] \
--conf "spark.driver.extraClassPath=/path_to/guice-4.0.jar" \
target/gcore-interpreter-1.0-SNAPSHOT-jar-with-dependencies.jar configrunner.json
```

Or set the `SPARK_HOME` variable with the command:

```
export SPARK_HOME=path_to_your_spark-2.4.6
```

and run:

```
spark-submit \
--class ggd.ERRunner \
--master local[2] \
--conf "spark.driver.extraClassPath=/path_to/guice-4.0.jar" \
target/gcore-interpreter-1.0-SNAPSHOT-jar-with-dependencies.jar configrunner.json
```

[6] https://github.com/smartdatalake/gcore-spark-ggd

The configrunner.json is a optional parameter pointing to a JSON file. The configrunner.json contains the information needed to connect by JDBC to Proteus and to the REST API of RAW.

After submitting the jar to the extended Spark project, the commands for the sHINER Component will appear as a command-line application. See section 6 for details on the configrunner.json and how to use the command-line application.

3.2 2 - Command-line using configuration file

Start the command-line application using the following commands.

```
./bin/spark-submit \
--class sHINERrunner \
--master local[2] \
--conf "spark.driver.extraClassPath=/path_to/guice-4.0.jar" \
target/gcore-interpreter-1.0-SNAPSHOT-jar-with-dependencies.jar
config.json comparison.json
```

Or set the `SPARK_HOME` variable with the command:

```
export SPARK_HOME=path_to_your_spark-2.4.6
```

and run:

```
spark-submit \
--class sHINERrunner \
--master local[2] \
--conf "spark.driver.extraClassPath=/path_to/guice-4.0.jar" \
target/gcore-interpreter-1.0-SNAPSHOT-jar-with-dependencies.jar
config.json comparison.json
```

After submitting the jar to the extended Spark project, the graph generation and the evaluation of the entity resolution will run according to the configuration files. This mode can also be used to run the graph pattern queries on the GGDs using JDBC connection. See more information about the configuration files and and how to set them on section 7.

3.3 3 - REST API

To run the project as a REST API use the following commands:

```
    ./bin/spark-submit \
    --class application.WebServer \
    --master local[2] \
path_to_GGDs_project/target/gcore-interpreter-ggd-1.0-SNAPSHOT-allinone.jar \
port_number /path/to/graph/datasets configrest.json
```

Or:

```
    spark-submit \
    --class application.WebServer \
    --master local[2] \
path_to_GGDs_project/target/gcore-interpreter-ggd-1.0-SNAPSHOT-allinone.jar \
port_number /path/to/graph/datasets configrest.json
```

After submitting the jar to Spark, a message that the server is online and its URL will show in the terminal. For more information on the routes and how to use the REST API see **??**. The configrest.json follows the same fromat as the configrunner.json in the command-line application.

4 Similarity Join operators

The Similarity Join operators have been added using the Spark Extensions API. This API allows you to extend the SparkSQL by including new Logical and Physical operators. By using this API we included Similarity Join operators for Jaccard and Edit similarity. These operators helps to run the Validation of the differential constraints in the GGDs faster than by comparing graph pattern query result to graph pattern query result.

You can run a Similarity Join in this project by using the following syntax:

```
SELECT * FROM table1 SIMILARITY JOIN table2 USING
        JACCARDSIMILARITY(table1.description, table2.description) < 0.8
```

To test the Similarity Join operators, use the following command:

```
./bin/spark-submit \
--class SimSQL.SimExample \
--master local[*] \
--num-executors 1 \
--driver-memory 512m  \
--executor-memory 512m  \
--executor-cores 1  \
path_to_GGDs_project/target/gcore-interpreter-ggd-1.0-SNAPSHOT-allinone.jar
```

And check the SimExample class on GitHub on more examples and information about the Similarity Join. If you are interested on how the Similarity Join operators were added to SparkSQL, please check the SimSQL package of the project. In there you will find the source code on how the parser was modified and how the logical and physical operators for Similarity Join were added.

5 Input Format

In this section we specify the input formats for both graph data and GGDs.

5.1 Graph Input Format

The graph input format follows the G-Core interpreter input format. Given the graph schema in Figure 18, the input format for the graph should have the structure in Figure 2:

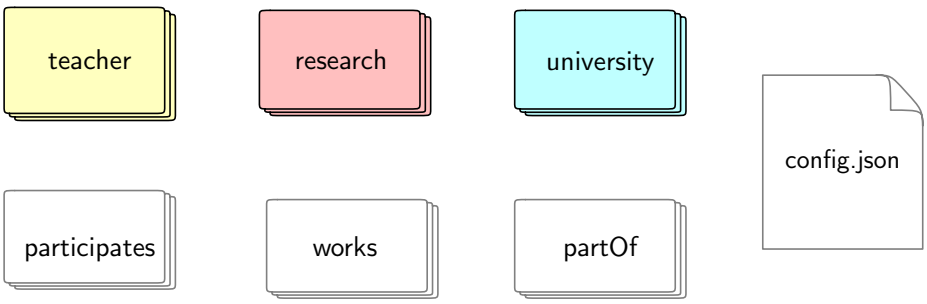

Fig. 2: Graph Input structure

Each vertex and edge label should have a folder in which each folder contains one or more JSON files with information on the vertices/edges of that specific label. For edges, besides the attributes for each edge it should also contain information on the source and target vertices by setting the attributes fromId and toId. For example, an edge instance of the label works should have the following JSON format:

```
{
"id": 1,
"since": "May 2019"
"fromId": 101
"toId": 102
}
```

In which fromId and toId refers to the source and target ids of the vertices it links. The `config.json` file is the configuration file of a graph and should declare information on the graph schema and also where the graph data is stored. The `config.json` is of the following format:

```
{
  "graphName": "research_graph",
  "graphRootDir": "defaultDB/research_graph",
  "vertexLabels": ["teacher","research","university"],
  "edgeLabels": [\participates", \works", \partOf"],
```

```
  "pathLabels": [  ],
  "edgeRestrictions": [
    {"connLabel": "participates",
      "sourceLabel": "teacher",
      "destinationLabel": "research"
    },{
      "connLabel": "works",
      "sourceLabel": "teacher",
      "destinationLabel": "university"
    },{
      "connLabel": "partOf",
      "sourceLabel": "research",
      "destinationLabel": "university"}],
  "pathRestrictions": []
}
```

The field graphName is the graph name that it should be saved and referred to in the dataset, graphRootDir is the path of the graph folder (folder which contains the config.json file and the vertices/edges folders), vertexLabels and edgeLabels are the labels present in the graph (name of the folders) and, edgeRestrictions gives information on the schema of the graph and which edge labels connect which vertices labels. With G-Core there is also the possibility of storing graph paths, however, graph paths are not currently being used for the ER component presented in this deliverable. For more information on how to store paths on G-Core refer to the G-Core language paper and the original G-Core interpreter project. Graphs stored in the defaultDB folder of the project are per default loaded into the component and can be used. It is also possible to load an external graph file in the project (see next Section). In case the user wants to save a graph that was loaded in the project database the output format of the graph is the same as the input format.

5.2 Graph Format - JDBC Query Execution

When using the JDBC, the sHINER component will use the configuration files of the datasets as a reference to understand which tables available by your JDBC connection are from which graph. So, the input format for the configuration files remains the same as when using SparkSQL but in this case there is no need of using the folders which contains data of each vertex/edge label. The support for JDBC connection in this project was designed for integrating with the SDL-Virt layer in the SmartDataLake project Architecture, in specific with the components RAW and Proteus. In Figure 4 we show an example on how to define a graph dataset using with Proteus and RAW and on **??** we show how the graph is loaded into the sHINER component using the JDBC connection.

When using the JDBC connection all graph pattern matching queries are executed by your JDBC component and not on SparkSQL. This is possible

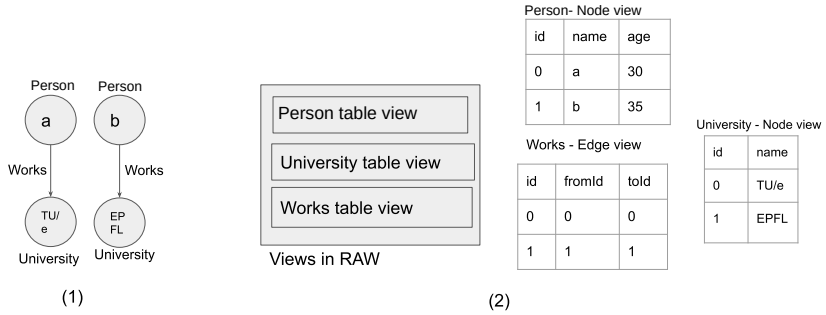

Fig. 3: Graph Dataset with Proteus and RAW

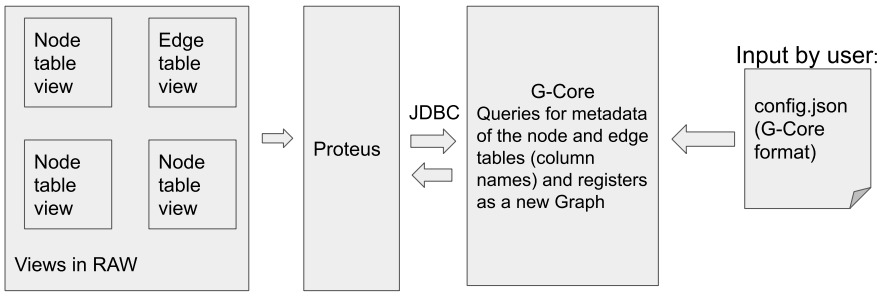

Fig. 4: Graph Load using JDBC

because the GCore Language Interpreter is responsible for rewritting GCore Queries into SQL queries. To run the queries using JDBC, we created a new "GCore Compiler" in which the queries are rewritten to standard SQL and run using the JDBC connection. Queries which standard sql does not support such as Path queries are executed using SparkSQL. The JDBC connection and the new "GCore Compiler" were specifically designed for integrating with the SDL-Virt layer in the SmartDataLake project Architecture

Observe on Figure 5 how graph pattern match queries are executed using JDBC.

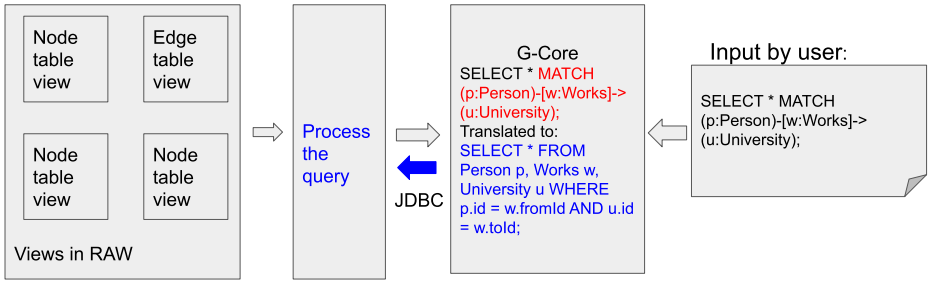

Fig. 5: Graph Query JDBC Execution

5.3 GGDs Input Format - Command-line Application

The input format for a Graph Generating Dependencies is composed of mainly two types of JSON files. A set of JSON files with information on the GGDs and a configuration file used by the component to load the set of GGDs. Each Graph Generating Dependency is defined in one JSON file in the following format:

```
{
"sourceGP": [{
"name": "dbpediaURL",
"vertices":[
{
"label": "dbpedia",
"variable":"z"
}],
"edges": []
},
{
"name": "dbpediaURL",
"vertices":[
{
"label": " yago",
```

```
"variable": "x"
}],
"edges":[]
}],
"sourceCons": [{
"distance": "edit",
"var1": "x",
"var2": "z",
"attr1": "name",
"attr2": "name",
"threshold": 1,
"operator": "<="
}],
"targetGP": [{
"name": "dbpediaURL",
"vertices":[
{
"label": "dbpedia",
"variable":"z"
},{
"label": " yago",
"variable": "x"
}],
"edges": [{
"label": "sameAs",
"variable": "y",
"fromVariable": "x",
"toVariable": "z"
}]
}],
"targetCons": []
}
```

A graph generating dependency, as introduced earlier, is composed of a source graph pattern and source constraints and a target graph pattern and constraints. In entity resolution, the source graph pattern and constraints are treated as a condition to link matching entities in the target graph pattern. In this context, the JSON file for a GGD have mainly 4 properties: sourceGP, sourceCons, targetGP, targetCons which refer respectively to the source graph pattern, source constraints, target graph pattern and target constraints. The sourceGP is a list of graph patterns in which each graph pattern is defined by:

```
    [{
"name": "dbpediaURL",
"vertices":[
{
"label": "dbpedia",
```

```
"variable":"z"
}],
"edges": [{
"label": "sameAs",
"variable": "y",
"fromVariable": "x",
"toVariable": "z"
}]
}
```

In which name is the name of the graph in the database, vertices is the list of vertices in the graph pattern and the edges is the list of edges in the graph pattern. While the targetGP is a single graph pattern that contains which links should be added in the target graph. Some of the restrictions on the graph patterns are: (1) The variable names should be unique to each vertex/node unless it always refers to the same entity (vertex/edge) and (2) Disconnected vertices/edges should be declared as a second graph pattern in the list. The sourceCons and the targetCons are lists of constraints in which each constraint should be declared in the following format:

```
  [{
"distance": "edit",
"var1": "x",
"var2": "z",
"attr1": "name",
"attr2": "name",
"threshold": 1,
"operator": "<="
}]
```

This format would translate to the following constraint:$\delta_e dit(x.name, z.name) \leq 1$. Which means that the similarity according to the edit distance between the name attribute of the variable x (referring to a vertex/edge in the graph pattern) and the name attribute of the variable z. The currently implemented (dis)similarity measures in the component and the keywords to be used in the JSON files are:

- Edit distance - “edit”;
- Euclidean distance – “euclidean”;
- Differential distance – “diff”

The operator can be one of the following symbols: `"<"`, `">"`, `"<="`, `"=>"`, `"="` and `"!="`. Currently the similarity join algorithms for Jaccard Similarity and Edit Distance are only supported for the operators `"<"` and `"<="`. In case the constraint is of

the type x=y (when it means that variable x refers to the same entity than y) the constraint should be declared by using the "equal" keyword in the following format:

```
        [{
"distance": "equal",
"var1": "x",
"var2": "z",
"attr1": "",
"attr2": "",
"threshold": 0,
"operator": "="
}]
```

The second JSON is the GGDs configuration file. The configuration file is a file responsible for giving information to the ER component on which GGDs you want to apply. The JSON should be given in the following format:

```
    {
"path": "/home/user/Documents/ggds/"
"names": ["ggd1", "ggd2", "ggd3"]
}
```

The path field should contain the path to the folder where the GGDs files are stored locally, while the names is the list of the names of the GGD files. The GGDs configuration JSON file is the file that should be the input for GGDs in the component, according to the attributes in the configuration json the component loads automatically the defined GGDs in the specified path.

Examples of GGDs in the specified format are available in the GGDsInput folder of the GitHub repository.

5.3.1 GGDs Input Format - REST API To make easier for the user to submit a list of GGDs through the REST API, we changed the input format of the GGDs from different files to one single json file with an array of GGDs. For example:

```
    [{
"sourceGP": [{
"vertices": [],
"edges": []
}],
"sourceCons": [],
"targetGP ": [{
"vertices": [],
"edges": []
}],
```

```
"targetCons": []
}, {
"sourceGP": [{
"vertices": [],
"edges": []
}],
"sourceCons": [],
"targetGP": [{
"vertices": [],
"edges": []
}],
"targetCons": []
}]
```

6 Command-line application

The command-line application of sHINER was built as an extension of the original G-Core runner available in the G-Core original repository. The configrunner.json file is a parameter that contains information about how to connect to Proteus and RAW components of the SmartDataLake architecture. The configrunner.json has the following format:

```
    {
"query":"jdbc",
"jdbcurl": "jdbc:avatica:remote:url=http://proteus_server:port/serialization=PROTOBUF",
"username": "proteus_jdbc_username",
"pwd": "proteus_jdbc_pwd",
"rawoath" : "authentication token for RAW Labs Connection (used only in case the query m
"rawurl" :  "url to connect to RAW REST API",
"rawsave" : "save local for raw : (s3, hdfs, file)",
"rawpath" : "path for saving files from RAW using jdbc"
}
```

In which query is “jdbc” to indicate the JDBC connection to Proteus/RAW or “spark” to indicate that the graph data and graph pattern queries will be run using SparkSQL. The other fields are about the url in which Proteus and RAW are installed and also about the user’s credentials needed to access Proteus and RAW. One observation is the rawsave field which is the path of a s3 amazon bucket, a hdfs file system or a file location in which both sHINER and RAW have permission to write and retrieve information from. This field is needed in case the user wants to run the graph generation (ER process) of sHINER.

When running the sHINER component in command-line mode, the functions available for using are specified in Figure 6.

We will now go through each option in this panel.

1. \f Load Database and \j Load graph - As mentioned before, the default folder from which the application loads graphs is the project defaultDB

Fig. 6: sHINER Command-line Panel

folder. However, there are other ways to load a new graph into the running application by using the options \f and \j. The Load Database option you load multiple graphs into the application by specifying the root folder in which all these graphs are stored. The Load Graph option you can load one single graph by specifying the root folder that this graph is stored. Both options will identify and load automatically information from the config.json files.

2. \l Graphs in database and \d Graph information – Both functions will give information about which graphs are loaded in the application. \l option will show the names of the available graphs while the option \d graph-name will show information about the schema of the graph with the name passed as parameter.
3. \c Save Graph – This option will save the specified graph in the specified path. The saving format of this graph is the same as the input format. This is handful in order to reuse the saved graph with the application.
4. ; Execute query – Query the available graphs in database by using G-Core queries. The result will showed in the terminal.
5. \g Load GGDs – Load the GGDs by inputting the config file path. This function will load GGDs that are later used to run the ER Component.
6. \e Run GGD Component – As the name suggests, this is the main function to run the ER Component (GGD Component). Through this function it will trigger the Validation and Graph Generation processes of the GGDs. First it will run Validation algorithm to check if there are any GGD being violated in the data and if yes, which instances violates the data. Then, if a GGD is violated Graph Generation function runs in order to repair the GGDs by generating new nodes or edges (in the case of Entity Resolution mainly "same_As" edges). This process is repeated until there are no violated GGDs. The resulting graph with generating edges is the same graph as specified in the target GGDs constraints. The resulting graph can be stored using the SaveGraph option.

7. \q – Stop running the application.

7 Command-line using Configuration Files

The command-line application using configuration files is a script which will run the validation and graph generation process according to the values setted in the configuration file (config.json, first argument) and will evaluate the entity resolution results according to the comparison json files (comparison.json, second optional argument). In this section we describe both the config.json and the comparison.json files.

The config.json file has information which describes how to load the graph datasets, if the queries will run using sparksql or jdbc connection, etc. The config.json has the following format:

```
{
 "query":"jdbc",
 "database": "path to the folder which contains the graphs/contains the config files fo
 "ggds": "path to the GGDs (configuration file)",
 "jdbcurl": "jdbc connection url, if query mode is spark set this field as an empty str
 "username": "proteus_jdbc_username",
 "pwd": "proteus_jdbc_pwd",
 "verification": true,
 "savegraph": "path for saving the resulting graph from graph generation",
 "verificationpath": "results saving path",
 "rawoath" : "authentication token for RAW Labs Connection (used only in case the query
 "rawurl" :  "url to connect to RAW REST API",
 "rawsave" : "save local for raw : (s3, hdfs, file)",
 "rawpath" : "path for saving files from RAW using jdbc"
}
```

In which each field corresponds to:

- query - Mode in which the graph patterns will run, it can be either ”spark” for SparkSQL or ”jdbc” for Proteus connection
- database - path to the folder which contains the graphs/contains the config files for loading the graph datasets
- ggds - path to the GGDs (configuration file)
- jdbcurl - jdbc connection url, if mode is ”spark” se this field as an empty string ””.
- username - jdbc username, if mode is ”spark” se this field as an empty string ””.
- pwd - jdbc password, if mode is ”spark” se this field as an empty string ””.
- verification - boolean value true if there is a second input argument for matched ids (comparison.json) and false if no verification is needed.
- savegraph - path for saving the resulting graph from graph generation

- verificationpath - if the verification filed value is true then set a verification path to save the results, results will be saved according to the order it was declared in the ground truth file. If verification value is false then this field should be an empty string.
- rawoath, rawurl, rawsave and rawpath - fields about the configuration for RAW (same as in the command-line application mode)

The comparison.json files is responsible having information about how to analyze the results for entity resolution (number of matched objects according to a ground truth). The comparison.json file is an array of object in the following syntax:

```
[{
  "groundtruth": "groundtruth-csv-files-path",
  "sep": ",",
  "pattern": {
    "name": "AmazonGoogle",
    "vertices": [
      {
        "label": "ProductGoogle",
        "variable": "g"
      },
      {
        "label": "ProductAmazon",
        "variable": "a"
      }
    ],
    "edges": [
      {
        "label": "sameAs",
        "variable": "s",
        "fromVariable": "a",
        "toVariable": "g"
      }
    ]
  },
  "id1": "a$idAmazon",
  "id2": "g$idGoogle"
}]
```

In which,

- groundtruth - path to groundtruth csv file which contains information of which id matches to which id in the format id1,id2.
- sep - Separator character in the csv file
- pattern - Graph Pattern that contains the matched entities, for example (Google)-[sameAs]-¿(Amazon). This pattern will follow the same format as the GGD input format.

- id1 - name of the column/attribute in the graph pattern to be used to match as id1 in the ground truth files. Use the variable name of the vertex/edge that contains the attribute to be compared to the ground truth file. For example, consider that we need to compare idAmazon attribute of the vertex ProductAmazon then we should use its variable name a and the separator symbol '$' and then the attribute name: a$idAmazon.
- id2 - name of the column/attribute in the graph pattern to be used to match as id2 in the ground truth files. Use the variable name of the vertex/edge that contains the attribute to be compared to the ground truth file.

By running the sHINNER component using these configuration files, in case the verificationpath field in the config.json file is not empty, it will output the following results for entity resolution in a csv file in the following format:

```
"ground-truth", "true-pos", "false-pos", "true-neg", "false-neg"
"groundtruth-path", 0,0,0,0
```

In which the ground-truth field corresponds to the grountruth csv file path in the comparison.json and the other values can be easily used to calculate very used measures such as Recall, Precision and F-Measure. The execution time for the whole process will be printed on the command-line log.

8 REST API

The configuration file for the REST API follows the same format as the for the command-line application. In the following, we list the commands available in the sHINER REST API and its details. We can submit these commands from, for example, a python script using the commands:

```
   data = json.dumps(data)
#data is where the arguments for the command are listed
r = requests.post('http://host:port_number/function', data=data)
response = r.content()
```

In case it is a get type of request, line 2 should be substituted for:

```
 r = requests.get('http://host:port_number/function')
```

In this document, we separate the commands in GET request commands and POST request commands. The sHINER REST API supports both entity resolution functionalities and also functionalities involving G-Core queries.

8.1 GET Commands

The commands using GET requests do not require any arguments to run (or the arguments are explicit in the URL). In the following, we explain the sHINER REST API commands that use GET request.

`http://host:port_number/graphDB`

This command will return the names of the available graphs in the sHINER component.

`http://host:port_number/graphDB/<graph-name>`

To submit this command, substitute the ¡graph-name¿ in the URL with the name of a graph available in the component. This command will return a string in the JSON format with the schema of the submitted graph name. For example, by submitting the URL: `http://host:port_number/graphDB/Buy`. In which the graph Buy schema can be represented by the Figure 7[2][7].

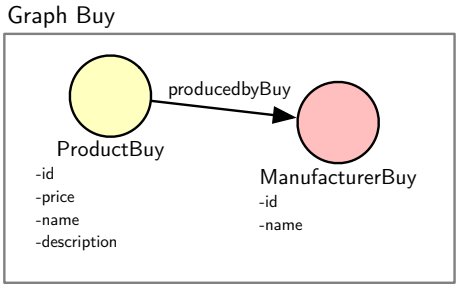

Fig. 7: Buy Dataset Schema Graph

The response content will contain the schema of the Buy graph in the node/link format used by javascript network libraries.

```
   {
"links": [{
"label": "producedbyBuy",
"property": ["fromId", "id", "label", "toId"],
"source": "ProductBuy",
"target": "ManufacturerBuy"
}],
"nodes": [{
"label": "ProductBuy",
"property": ["description", "id", "price", "label", "name", "idBuy"]
}, {
"label": "ManufacturerBuy",
"property": ["description", "id", "price", "label", "name", "idBuy"]
```

[7] Schema based on the Abt-Buy dataset provide by `https://dbs.uni-leipzig.de/research/projects/object_matching/benchmark_datasets_for_entity_resolution`

```
}]
}
```

`http://host:port_number/ggds/getGGDs`

This command retrieves a single JSON containing the GGDs set in the sHINER component.

`http://host:port_number/ggds/runER`

The runER command will start running the entity resolution by using the GGDs set in the component by the user. When the process is over it will return the name of the resulting graph and information about the generated edges in the following format:
//TODO OUTPUT FORMAT HERE

8.2 POST Commands

The commands using POST request type require arguments to run. The sHINER REST API commands that use POST request type are explained in the following.

`http://host:port_number/ggds/setGGDs`

This command is used for sending and setting GGDs that will be used for entity resolution in the sHINER component. While in the command line application of sHINER there is a configuration file that indicates all GGDs in separate files, in order to make the submission from a REST API easier, this command accepts an array of GGDs in a single JSON file in the format:

```
[{
"sourceGP": [{
"vertices": [],
"edges": []
}],
"sourceCons": [],
"targetGP ": [{
"vertices": [],
"edges": []
}],
"targetCons": []
}, {
"sourceGP": [{
"vertices": [],
"edges": []
}],
"sourceCons": [],
"targetGP": [{
```

```
"vertices": [],
"edges": []
}],
"targetCons": []
}]
```

More information about the GGDs input format is available in deliverable 3.2. The setGGDs command will return a message "GGDs set!!" if the GGDs were set correctly in sHINER or a message "GGDs were not sent correctly" if the GGDs were not correctly set in sHINER.

The following commands are for using G-Core to query the graphs available in sHINER. These commands can be used to query the resulting graph after running the entity resolution component.

```
http://host:port_number/gcore/select
```

Command to submit a SELECT type of query to G-Core. The argument to submit this command to the server is a json specifying the query and the number of results needed ("limit"), if the user is interested in retrieving all the results the "limit"value should be set to -1. For example:

```
{\query":\SELECT * MATCH (a:ProductBuy) ON Buy", \limit":50}.
```

SELECT queries on G-Core return a tabular view of the specified matches of the query. The response to this command is a JSON file with the results of the query.

```
http://host:port_number/gcore/construct
```

Command to submit a CONSTRUCT type of query to G-Core. The argument to submit this command to the server is the same format as for SELECT queries, a json in the format:

```
{\query":\CONSTRUCT (a) MATCH (a:ProductBuy) ON Buy", \limit":10}.
```

The answer to CONSTRUCT queries is a new graph built according to the specified graph pattern in the query string. In this case, since the answer to the query is a graph, the response to this command is a graph following the node/link format (same format as retrieving the schema of a graph)

```
http://host:port_number/gcore/select-neighbor
```

Command to return a tabular view of the neighbors of a specified node. The argument to this command is a JSON file with the specifications of the node you are interested in in the following format:

```
{
"nodeLabel": "ProductBuy",
"id": "1",
"edgeLabel": "",
"graphName": "Buy",
"limit": 1
}
```

In which nodeLabel and id is the label and id of your reference node, respectfully. The argument edgeLabel is the label/type of neighbors you want to retrieve, (if there is no specific type of neighbor it should be left in blank), graphName is the name of the graph you want to search and limit is the number of neighbors you want to retrieve.

Similar to the SELECT G-Core query, this command will return as response a JSON file containing information about neighbors of the specified node.

`http://host:port_number/gcore/construct-neighbor`

The construct-neighbor command uses the same argument as the select-neighbor command, however the response is a graph containing all the neighbors in the node/link format.

`http://host:port_number/gcore/createGraphs`

This function has the following input format:

```
{
\graphName":"name"
\unionGraphs":[\graph1", \graph2"]
}
```

This function will register a new graph with the name declared in the graphName attribute of the request which contains the information of graph1 union with the information of graph2. While logically, sHINER will treat the created graph as a new graph, in practice it will only create a new configuration file which points to the data of graph1 and graph2. If the user wants to perform entity resolution on two different datasets/graphs, he can create a new graph with this function and use it as a target graph to indicate where the links will be created. The response format for this command is a string message: "The resulting graph name is: name", if the command is successful or "Could not create the target graph" if an error occurred.

`http://host:port_number/gcore/dropGGDs`

This command deletes the generated tables and GGDs responsible for generating these tables from sHINER. This is useful if the user wants to refine its results and delete previous generated links. The request format for this command is:

```
[{
\name":"same_As",
\number":100,
\ggd":0,
\graphName":"AbtBuy",
\id":0
}]
```

Where, name is the label that the user wants to delete, the number is the number of instances that were generated, the GGD is the index of the GGD in the set, the graphName is the name of the graph the user wants to delete the label from and id is the index in the list of the request array.Observe that the request format in this command is the same as the resultInfo from the get /runER command. The response is a message: "The selected GGDs were deleted!", if there is no error in deleting the GGDs and its generated labels or "There was an error on removing the selected GGDs" in case an error occurred.

9 sHINER Extension - Hierarchical Graph Visualization

The SDL-Vis component of the SmartDataLake architecture includes a panel on visualization Graph dataset data using a Hierachical Clustering technique in the back-end. This allows the user to visualize the graph data in different levels according to the region (cluster) the user intends to visualize. To build the back-end of this visualization panel we extended the sHINER component using the Spark Machine Learning library and added new commands to the REST API to support this visualization in SDL-Vis. In this section, we describe the REST API calls for the hierarchical graph visualization functionalities. All commands are POST type commands.

`http://host:port_number/graphvis/initialvis`

Command for sending the initial parameters and generating the first visualizations based on clustering. The expected parameter is a JSON of the format:

```
{
  "attributes": [
    "a$sepal_length",
    "a$sepal_width",
    "a$petal_length",
    "a$petal_width"
  ],
  "graph": {
    "name": "Iris",
    "vertices": [
      {
        "label": "iris",
```

```
        "variable": "a"
      },
    ],
    "edges": []
  },
  "algorithm": "kmeans",
  "feature": "none",
  "algoParams": "{\"k\": 3, \"reps\": 5, \"algorep\": \"topk\"}"
}
```

1. Attributes are the attributes in which are important for the hierarchical visualization and the attributes in which the clustering methods will be performed over. Include the variable name defined in the graph pattern and the separator "$" so that the sHINER component can easily understand which node or edge attribute it is referring to.
2. Graph is the graph pattern which the user will get information from to do the clustering. Each match of the graph pattern will be treated as an feature vector in the clustering/ building hierarchical representation
3. Algorithm: the algorithm to be used as the clustering method
4. Feature: the algorithm to be used in transforming the attributes in a feature vector, if "none" it assumes that the attributes are already numerical values that can form a feature vector.

Response format:

```
    {
   "graph": {
       "links": "[]",
       "nodes": "[{\"a$petal_length\":\"6.4\",\"a$id\":131,\"label\":\"iris\",\"cluster\
},
    "similarity":"[[0.39234, 0.498023, 1.291083], [1.230990, 2.1389830], [2.23090]]"
    "visid": "UboFDbFWTLJIcJA"

}
```

Graph Link format for the graph patterns chosen as representative patterns and similarity matrix (upper triangular format as the distance used is euclidean distance and is symmetric) with the distances from a graph pattern to other graph pattern. Using the rowId as an key

```
http://host:port_number/graphvis/nextlevel
```

Command for sending the parameters expected to get the next level of points of a determined cluster and the closest points of the other clusters to visualize. The expected parameter is a JSON of the format:

```
{"clusterid":1, "level":4, "numpoints":3,"visid": "UboFDbFWTLJIcJA", "id":99}
```

1. Clusterid is the id of the cluster that the user is interested in visualizing/-zooming in
2. Level: is the level of graphs to be retrieved. For example if the initial level is 1, then to retrieve the level 2 of the representation use "level":2. Using this as a parameter to support cases of zooming out or zooming into the same level but using other cluster id.
3. Numpoints: Number of points/graph patterns to be retrieved from clusters with different id from the parameters clusterid.
4. Visid: The id of the initial visualization (similar to a session id, used as key to get the information from the hierarchical representation created in the initialVis request)
5. Id: Id of the Response format. Use cluster to identify the cluster that each graph pattern belongs to.


```
 {
"graph": {
    "links": "[]",
    "nodes": "[{\"a$petal_length\":\"4.1\",\"label\":\"iris\",\"cluster\":1,\"a$sepal
},
"visid": "UboFDbFWTLJIcJA"
}
```


10 Proteus/RAW Connection

In this section we provide an Overview of the Workflow of the main algorithms of the GGDs using the connection to Proteus and RAW.

10.1 Load Graph Data from RAW

To register a graph into sHINER to be used in the other functionalities the data analyst first needs to create views in RAW in which each view would represent one label/type of vertex or edge. Having created the views, the user loads the config.json for each graph dataset into sHINER (see deliverable D3.3 for more details on the configuration file input format). The Figure 8 shows an example of declaring graph views in RAW and the general process of registering a graph into sHINER using RAW and Proteus.

Observe that the graph input format does not change when compared to using the standalone version of the component that loads graph data using JSON files. The difference is that in this case the data is only registered on sHINER using a query to retrieve metadata of the views and is only actually retrieved an saved into sHINER component memory when a query is issued. The Figure 9 shows an overview of the process according to the different modules involved in this task.

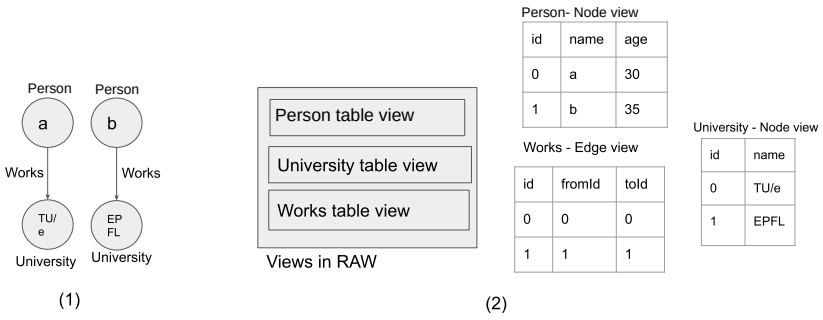

Fig. 8: Graph Dataset with Proteus and RAW

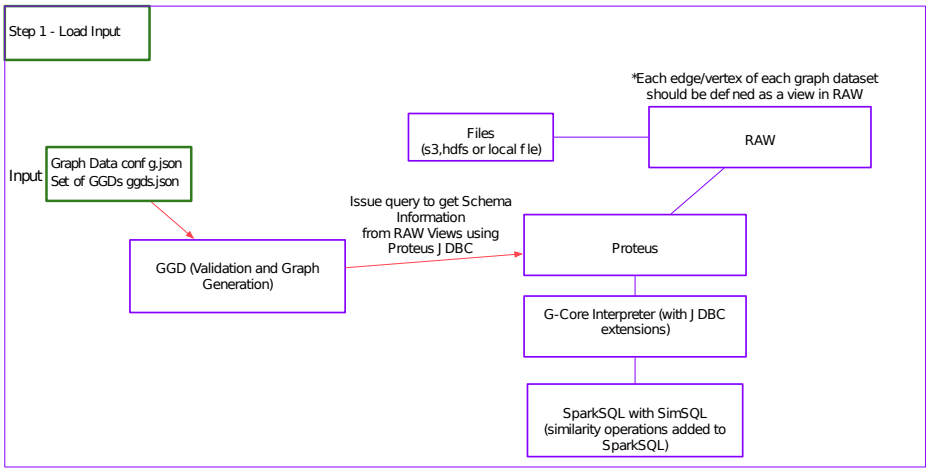

Fig. 9: Load a graph dataset using Proteus/RAW

10.2 Validation Algorithms

After registering all the graphs into sHINER the next step to run the Entity Resolution application is to run the Validation Algorithm. This algorithm is responsible for checking which of the GGDs are violated, in the case of ER, which data are candidates to be linked/deduplicated. To run the validation algorithm we need to run graph pattern matching queries in our graph. When using the JDBC connection all graph pattern matching queries are executed by your JDBC component and not on SparkSQL. This is possible because the GCore Language Interpreter is responsible for rewritting GCore Queries into SQL queries.

To run the queries using JDBC, we created a new "GCore Compiler" in which the queries are rewritten to standard SQL and run using the JDBC connection. Queries which standard sql does not support such as Path queries and similarity queries are executed using SparkSQL. The JDBC connection and the new "GCore Compiler" were specifically designed for integrating with the SDL-Virt layer in the SmartDataLake project Architecture.

See subsection 5.2 for details on how we translate the graph pattern queries to SQL and run it on Proteus.

Observe in Figure 10 the overall workflow of the Validation Algorithm using the JDBC connection to Proteus and the graph views in RAW.

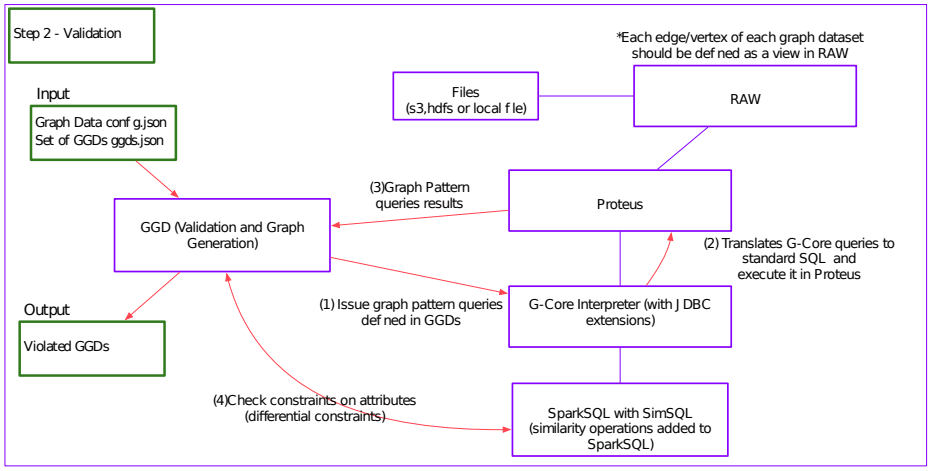

Fig. 10: Validation algorithms using Proteus/RAW

10.3 Graph Generation Algorithm

At last, the next step to run the entity resolution is to run the Graph Generation algorithm which will create links between the matched data.

This algorithm runs in a similar way to the Validation algorithm as we use the new G-Core compiler to translate the queries to standard SQL and run all the graph pattern queries on Proteus. The main difference is that when generating new links/edges, these links need to be stored and accessed as view in RAW so the user can be able to run queries and verify the results (the created edges) using also the JDBC connection. In conclusion, for each new type of generated edge we need to create a view on RAW of this new edge so it can be used for querying later.

To create a new view in RAW, we first save the generated edges data in a JSON file in a file system that the user configured when starting sHINER, it could be a HDFS file system or even just a local path in the machine the user is running sHINER on, the only detail is that the user needs to make sure that both sHINER and RAW installation can access this files system/path. Next, we use RAW's REST API[8] to create a view using the saved JSON and finally we updated the configuration of the graph this new type of edge is now part of inside sHINER.

The Figure 11 shows an overview of the workflow of the graph generation algorithm using Proteus and RAW.

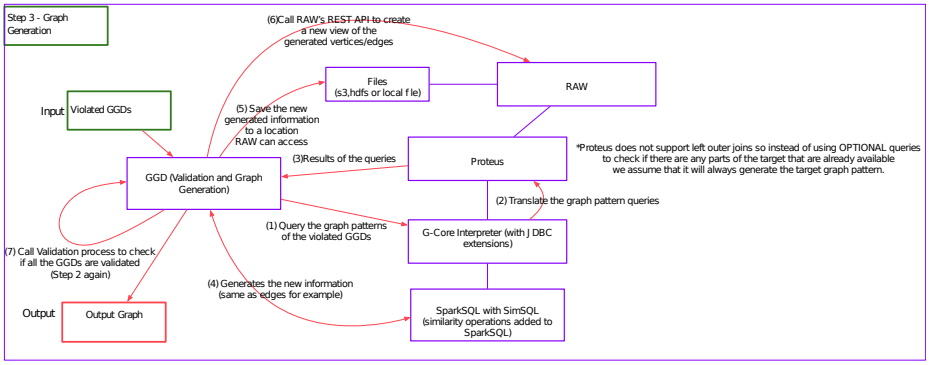

Fig. 11: Graph Generation algorithm using Proteus/RAW

References

1. R. Angles, M. Arenas, P. Barcelo, P. Boncz, G. Fletcher, C. Gutierrez, T. Lindaaker, M. Paradies, S. Plantikow, J. Sequeda, O. van Rest, and H. Voigt. G-core: A core for future graph query languages. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, page 1421–1432, New York, NY, USA, 2018. Association for Computing Machinery.
2. H. Köpcke, A. Thor, and E. Rahm. Evaluation of entity resolution approaches on real-world match problems. *Proc. VLDB Endow.*, 3(1–2):484–493, Sept. 2010.

[8] https://docs.raw-labs.com/latest/usage/rest.html

3. L. C. Shimomura, G. Fletcher, and N. Yakovets. Ggds: Graph generating dependencies. In *Proceedings of the 29th ACM International Conference on Information Knowledge Management*, CIKM '20, page 2217–2220, New York, NY, USA, 2020. Association for Computing Machinery.