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). This project needs the spark-2.4-Sim also available in the SmartDataLake repository which contains these similarity operators[5]. The Figure 1 shows the general architecture of the sHINER component and how it was implemented.

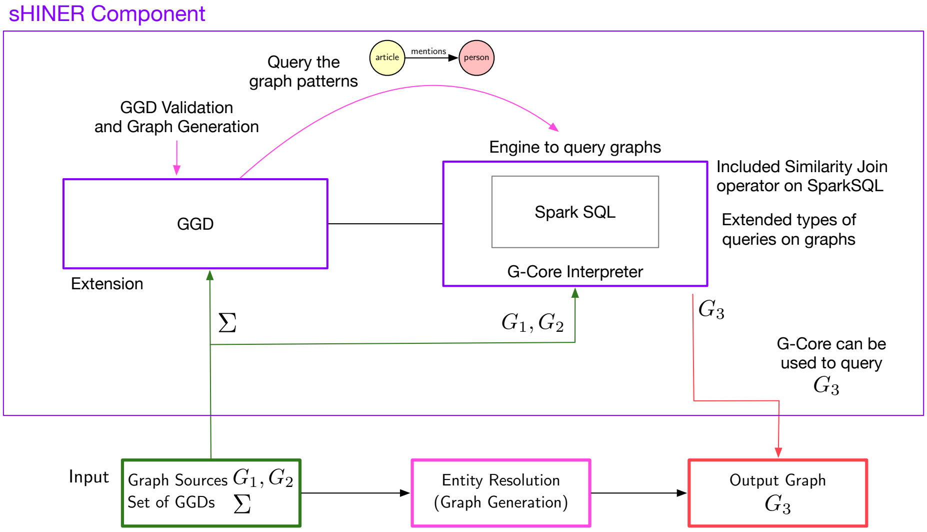


Fig. 1: Architecture of sHINER

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. Guice library jar file[6]

[5] https://github.com/smartdatalake/spark-2.4-sim
[6] 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[7] component and the Spark-Sim[8] project in the SmartDataLake GitHub repository. Since the sHINER (GGDs component) needs the similarity join operator for the Validation of the GGDs, an standard Spark instance will not be able to run correctly the validation of the GGDs.

To build the Spark-Sim instance, use the same instructions as building Spark from source code:

```
git clone https://github.com/smartdatalake/spark-2.4-sim.git

cd spark-2.4-sim

build/mvn -DskipTests clean package
```

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

```
./bin/spark-submit \
--class org.apache.spark.examples.sql.SimJoinExample \
--master local[*] \
--num-executors 1 \
--driver-memory 512m  \
--executor-memory 512m  \
--executor-cores 1  \
examples/target/original-spark-examples_2.11-2.4.7-SNAPSHOT.jar 10
```

For using this Spark instance in cluster mode, use the instructions available in Spark website/repository.

Once Spark-Sim is built, the next step is build the sHINER Component. To build the sHINER use the following instructions:

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

cd gcore-spark-ggd

mvn install
```

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

[8] https://github.com/smartdatalake/spark-2.4-sim

When the project is built, the sHINER component REST API can be started with the following command:

```
./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
```

Or set the SPARK_HOME variable with the command:

```
export SPARK_HOME=path_to_your_spark-2.4-sim
```

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
```

After submitting the jar to the extended Spark project, the commands for the sHINER Component will appear as a command-line application.

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
```

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
```

After submitting the jar to the Spark-Sim project, a message that the server is online and its URL will show in the terminal.

4 Input Format

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

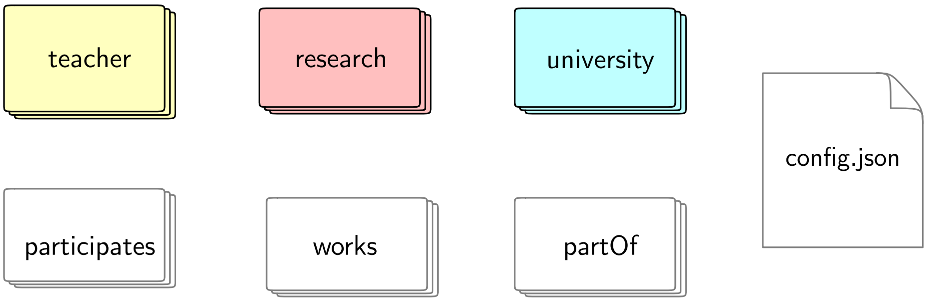


Fig. 2: Graph Input structure

4.1 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:

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.

4.2 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.

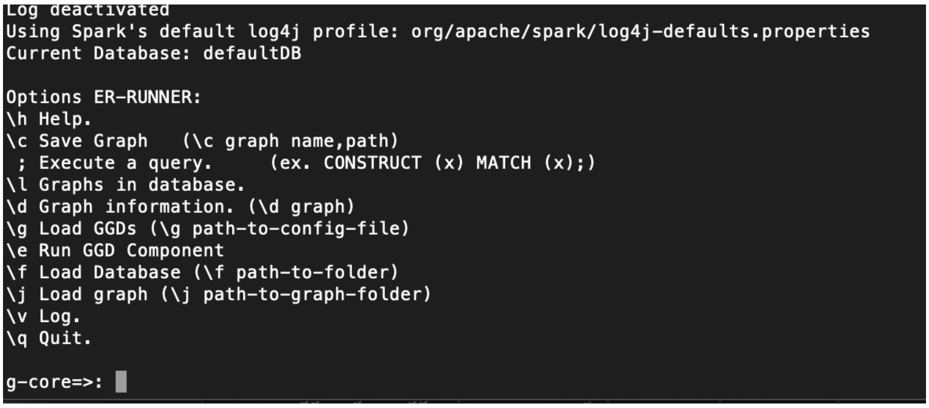
Examples of GGDs in the specified format are available in the GGDsInput folder of the GitHub repository.

4.2.1 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": []
}]
```

5 Command-line application

When running the sHINER component in command-line mode, the functions available for using are specified in Figure 3.



```
Log deactivated
Using Spark's default log4j profile: org/apache/spark/log4j-defaults.properties
Current Database: defaultDB

Options ER-RUNNER:
\h Help.
\c Save Graph   (\c graph name,path)
 ; Execute a query.     (ex. CONSTRUCT (x) MATCH (x);)
\l Graphs in database.
\d Graph information. (\d graph)
\g Load GGDs (\g path-to-config-file)
\e Run GGD Component
\f Load Database (\f path-to-folder)
\j Load graph (\j path-to-graph-folder)
\v Log.
\q Quit.

g-core=>: █
```

Fig. 3: sHINER Command-line Panel

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 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.

6 REST API Routes

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://localhost: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://localhost: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.

6.1 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://localhost:port_number/graphDB
```

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

```
http://localhost: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://localhost:port_number/graphDB/Buy`. In which the graph Buy schema can be represented by the Figure 4[2][9].

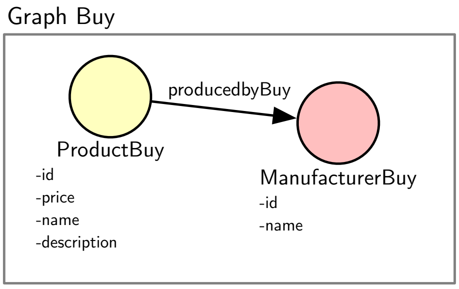


Fig. 4: Buy Dataset Schema Graph

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

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

```
   {
"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"]
}]
}
```

```
http://localhost:port_number/ggds/getGGDs
```

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

```
http://localhost: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.

6.2 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://localhost: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://localhost: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 string specifying the query. For example: `"SELECT * MATCH (a:ProductBuy) ON Buy"`.

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://localhost: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 a string specifying the query. For example: “CONSTRUCT (a) MATCH (a:ProductBuy) ON Buy”.

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://localhost: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://localhost: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.

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.
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.