Tampere University

Nguyen Quang Tung

SUITABILITY OF GRAFANA AND KIBANA FOR VISDOM-TYPE VISUALIZATIONS

Bachelor's Thesis

Faculty of Information Technology
and Communication Sciences

Kari Systä

April 2020

THIS PAGE IS INTENTIONALLY LEFT BLANK

ABSTRACT

Nguyen Quang Tung: Suitability of Grafana and Kibana for VISDOM-type visualizations
Bachelor's Thesis
Tampere University
International Bachelor's Degree Program in Science and Engineering
April 2020

This paper aims to find out whether it is possible to convert the existing visualization dashboards of the VISDOM-project, that is using the d3.js library for its visualization component, to a more recent software (Kibana or Grafana). VISDOM-project is a relatively new innovation in the field of data visualization, with a focus on providing visualizing tools for software development projects. The research in this paper was done by first obtaining the data that is needed to be visualized and examining it to find out whether there needs to be any pre-processing, which will be addressed during the implementation. Afterwards, a comparison will be made between potential data visualization tools to single out the best candidate to be tested, and its requirements and functionalities will be thoroughly studied. Finally, all the steps necessary for the data as well as the tool will be performed, until the visualizations are yielded as a result. The result is then compared to the original figures taken from VISDOM-project to determine if the resulting implementation are satisfactory. Ultimately, a final conclusion will be drawn by means of comparison between the original and replicated visualizations, combined with various other factors such as complexity, replicability and how manual the new implementation is. For this research, it has been determined that it is only possible to implement a part of the required visualizations, and thus the best option is to stick to the current d3.js implementation until further developments are made in the field of data visualization.

Keywords: Data Visualization, Grafana, Kibana, PostgreSQL, VISDOM-project

PREFACE

I would like to thank Assistant Prof. Kari Systä for supervising my work and helping me in a variety of ways, and Mr. Ville Heikkilä for helping me with the visualization part. I would also like to thank Ms. Mervi Miettinen for giving instructions on information retrieval, as well as Assistant Prof. Laeticia Petit for organizing this year's Bachelor's Thesis Seminar and motivating me towards finishing my thesis.

Tampere, 27 April 2020

Nguyen Quang Tung

CONTENTS

LIST OF FIGURES

LIST OF PROGRAM SNIPPETS

LIST OF SYMBOLS AND ABBREVIATIONS

B.C	Before Christ
DevOps	Software development (Dev) and IT operations (Ops)
JSON	JavaScript Object Notation
AWS	Amazon Web Service
DOM	Document Object Model
ELK	The Elasticsearch – Logstash – Kibana stack
SQL	Structured Query Language
IT	Information Technology

1. NTRODUCTION

VISDOM-project is a joint effort of individuals and organizations from all over Europe, aiming to make a breakthrough in the field of data visualization by introducing a novel approach in viewing data; that is, to utilize data from multiple sources during the development of DevOps projects and the like, providing developers with occasional “health checks” for the software [1].

The code of VISDOM-project is based on the implementation of an earlier, similar project called the N4S project, with a visualization system utilizing the d3.js library on top (as seen from the Github page of the application implementation [2]). While that has still been going relatively well, the emergence of newer, more modern means of visualizing data such as Grafana and Kibana has prompted numerous questions on whether the project should switch to one of those visualization software due to the apparent convenience and ease of design; it takes hundreds of lines of code to produce a single visualization using the d3.js library, whereas Grafana requires but a few lines of data query to get the needed data, then a little option tweaking to get the right kind of graph to display. With that in mind, the writer was tasked to find out if the migration of software can be done with ease and convenience, particularly by re-implementing the available visualizations from VISDOM-project using only one tool. The detailed description of the task at hand, as well as steps towards solving it and the ultimate conclusion will be mentioned in the following chapters.

In chapter 2, all relevant concepts regarding this problem will be explained in detail, while chapter 3 attempts to use a visualization alternative for VISDOM-project’s tasks. Afterwards, chapter 4 will take the results from the previous part to analyze and give a verdict on the suitability compared to the current d3.js implementation. Last but not least, chapter 5 will mark the conclusion of this paper.

2. THEORETICAL BACKGROUND

This chapter focuses on providing the theoretical knowledge needed to grasp further sections of the paper.

2.1 A brief history of data visualization

Data visualization, in its most primordial form, has its origins dated thousands of years ago; going back to as far as 200 B.C. with the use of coordinates in maps [3]. However, the kind of data visualization that we are familiar with only made its appearance few centuries ago.

The visualization forms that are close to that of the current days only emerged from the late 18th century, when William Playfair, a Scottish social scientist, made various adjustments to things like bar or pie charts, which had only been invented at the start of the same century [4]. Half a century later, however, marks the period of exponential growth in the variety of visualizations, including the likes of histograms or even atlases, due to the need to picture data in a multitude of popular fields, both scientific and societal [3].

With that being said, the importance of data visualization was not recognized until the 20th century, when the first academic course regarding the topic was introduced at Iowa State University in 1913 [4]. Afterwards, even though the variety of visualizations did not witness such substantial increase like before, it was a breakthrough in the field of software engineering that gave rise to the popularity of data visualizing: The personal computer was introduced in 1977. With the computer handling most of the manual and meticulous work such as drawing figures and plotting data, data visualization has never been this quick and easy. Ever since then, rapid developments have been recorded in each decade, or even each year, starting with the introduction of Microsoft Excel in 1987 [5]. Nowadays, data visualizations exist in all facets of the world, from tables and mathematical graphs in textbooks to analyzing huge datasets in Amazon Web Service or Azure:

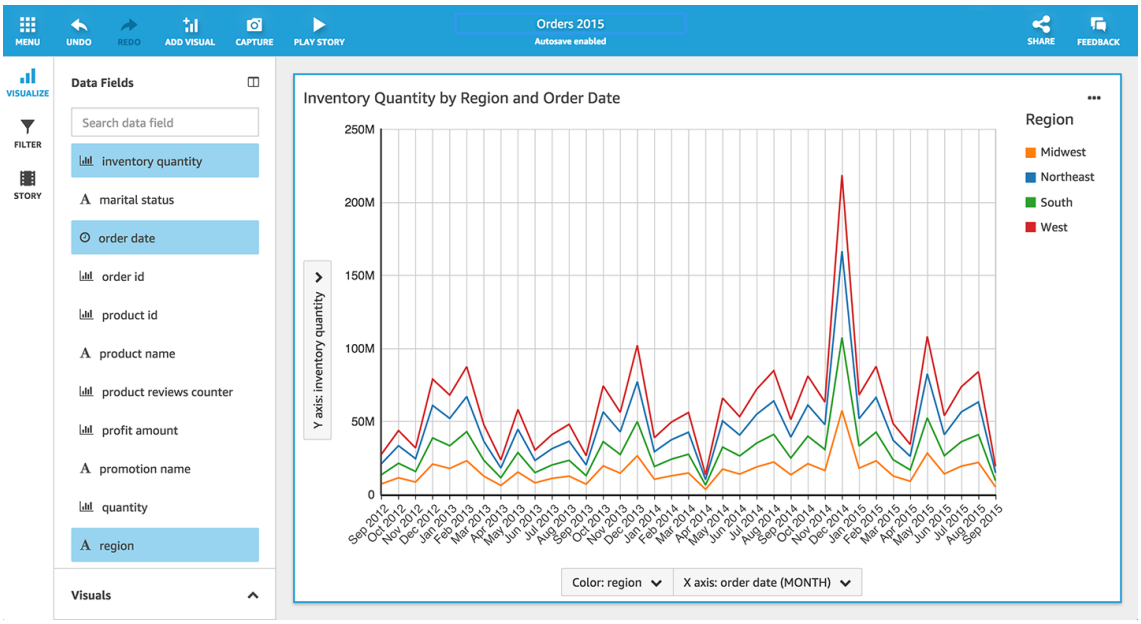

Figure 1. AWS visualizing hundreds of millions of entries of data [6]

One of the advantages of visualizing data lies in the biological nature of a person. Human eyes are more likely to notice differences in colors and shapes rather than number; the concept of drawings and sketches have existed long before numbers were invented. Additionally, it is generally easier for people to detect trends and outliers in graphs than it is in tables of numbers [7].

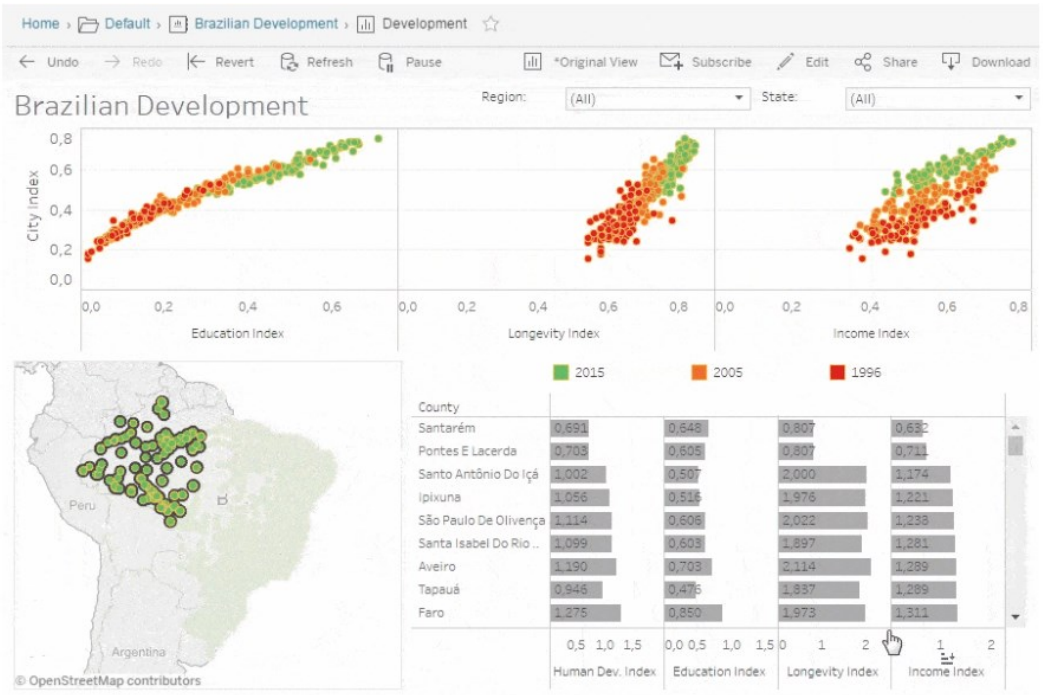

Figure 2. Graphs make it much easier to detect trends than just numbers (on the bottom right) [7]

The increasing popularity of data visualization throughout history has gathered the attention of numerous companies of all sizes, from the likes of Amazon (AWS), Windows (Microsoft Office) or Google (Maps) to small organizations like the VISDOM-project.

2.2 About the VISDOM-project

As discussed above, VISDOM-project utilizes data visualization to bring forth solutions to customers. Its specialization is in the software engineering field, where occasional “health checks” are needed to ensure that the software is operating as intended. One of VISDOM-project’s innovations include the integration of visualization components from multiple tools into a unified view originally created to save developers time from repeatedly going through figures after figures.

People from VISDOM-project see that the process of creating and maintaining a software is comparable to many typical biological functions of a human being. For example, repeating periods within the production of software such as sprints or delivery cycles can be interpreted similarly to the periodicity of an EKG (electrocardiography) curve. Likewise, it has been found out that some phase can represent the changes in blood pressure or heart pulse. This analogy can help people understand the details in the graphs more easily and make the purpose of each visualization stand out even more.

Last but not least, VISDOM-project offers different types of dashboard for each stakeholder, visualizing only what the person needs to see. This eases the burden of navigation between graphs for everyone, allowing for faster problem diagnosing and treatment [1].

With a large number of tasks that needs to be accomplished, VISDOM-project has to choose a visualization platform competent enough to address all of the problems. With that in mind, d3.js (Data-Driven Documents) has been chosen as the primary tool for the project [8]. Written using JavaScript, d3.js is a library comprising of a multitude of features for working on data, with data visualization at its core. It allows users to put data into the DOM and execute various actions on it, including visualizing the data and much more [9].

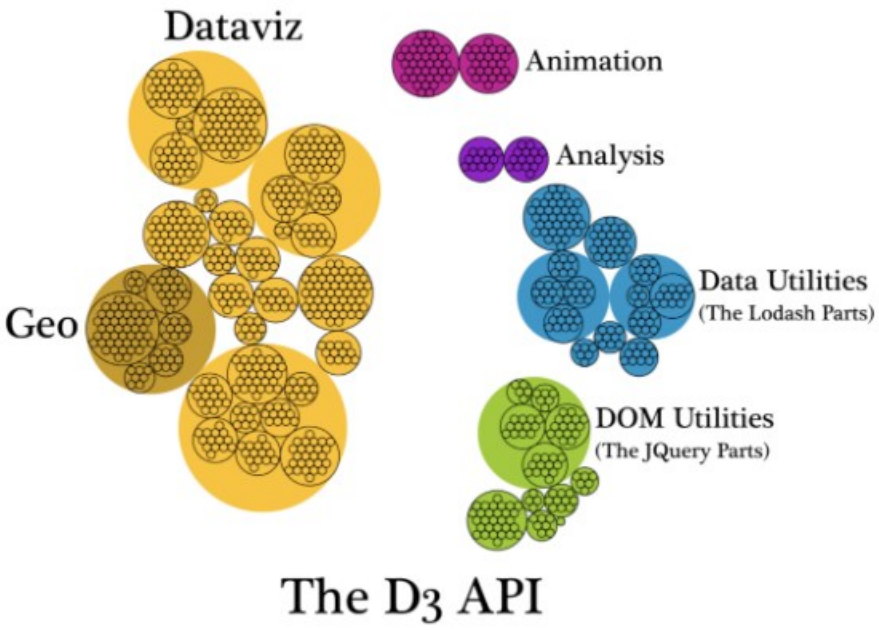

Figure 3. d3.js is more than just a data visualization library [10]

The variety of features as well as the depth of its API has proven to be sufficient for VISDOM-project to rely on for the moment, but the project is always on the lookout for more efficient and convenient tools to visualize its data. After extensive efforts being made, the search was narrowed down into two best candidates: Grafana and Kibana.

2.3 On Kibana and Grafana

In addition to offering a wide range of visualization options, both Kibana [11] and Grafana [12] are also capable of combining different types of visualizations into an object called a “dashboard”, and some examples can be seen below:

Figure 4. An example Grafana dashboard from a sample dataset [13]

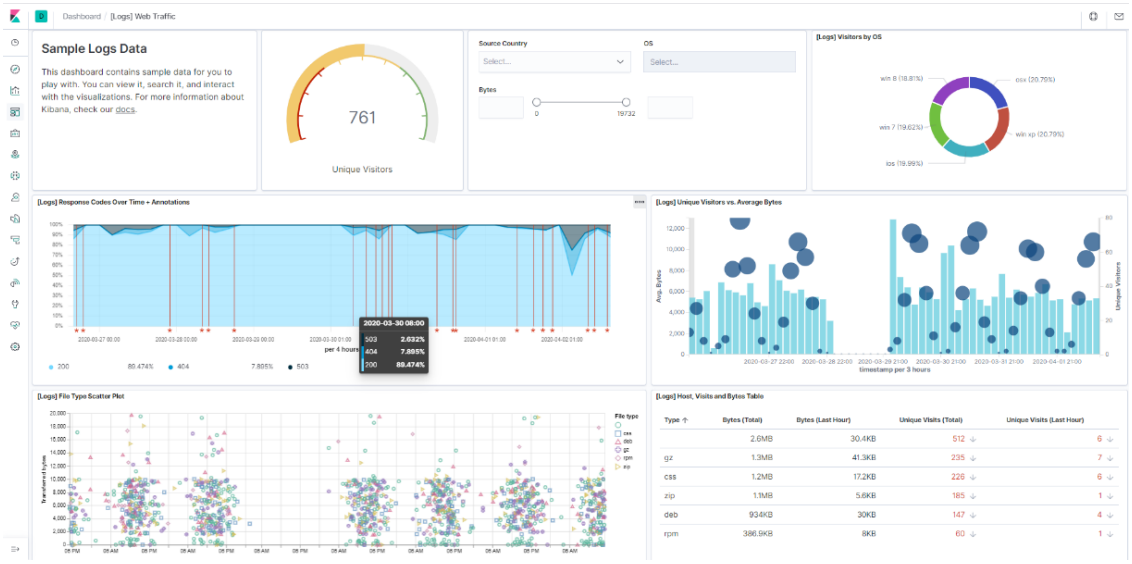

Figure 5. An example Kibana dashboard from a sample dataset (part 1) [14]

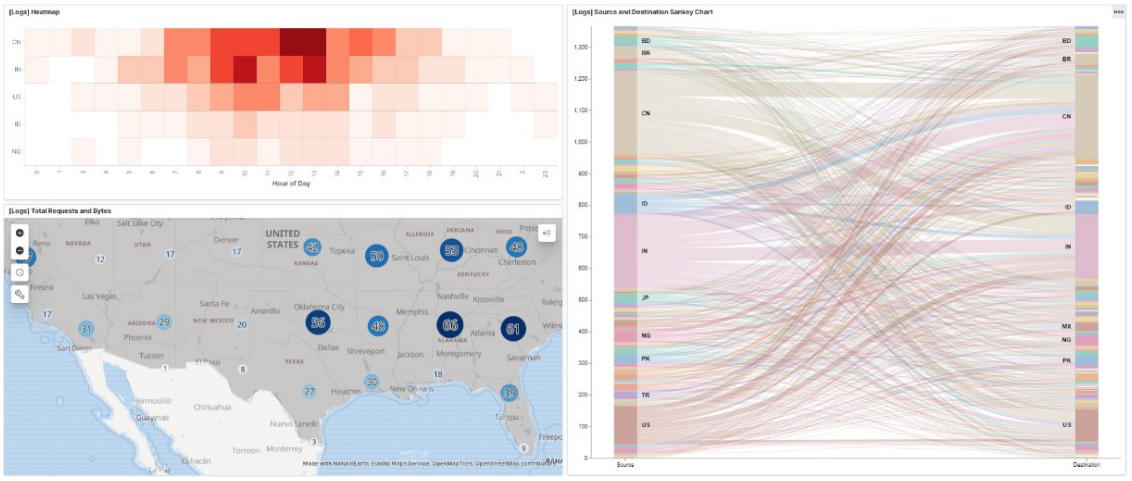

Figure 6. An example Kibana dashboard from a sample dataset (part 2) [14]

For both of the above-mentioned software, a typical data visualization process usually consists of the following tasks:

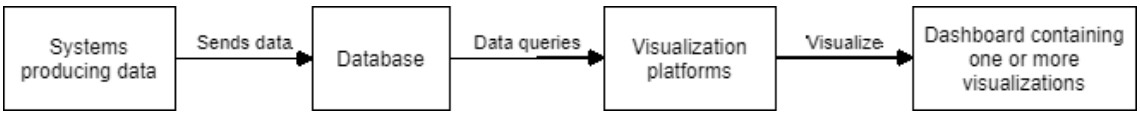

Figure 7. Data visualization workflow

While Grafana accepts a large number of popular databases, Kibana, being the “K” of the ELK (Elasticsearch – Logstash – Kibana) stack, restricts itself to Elasticsearch, which is not just a database. This means that all data from popular databases must be transferred to the Elasticsearch platform before it can be used in Kibana, which can be seen in the figure below:

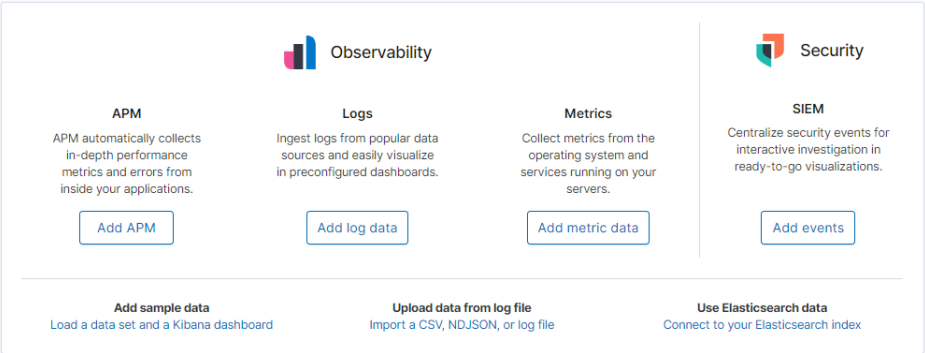

Figure 8. Limited options for Kibana data sources [14]

This brings about a particular disadvantage of Kibana, where in order to utilize it, one has to be familiar with Elasticsearch as well, whereas knowledge in typical databases such as SQL-related ones or Amazon Web Service's CloudWatch, etc. is sufficient for using Grafana; in fact, Grafana also supports the use of Elasticsearch as a database source, and it can be clearly seen from the selections of data sources below:

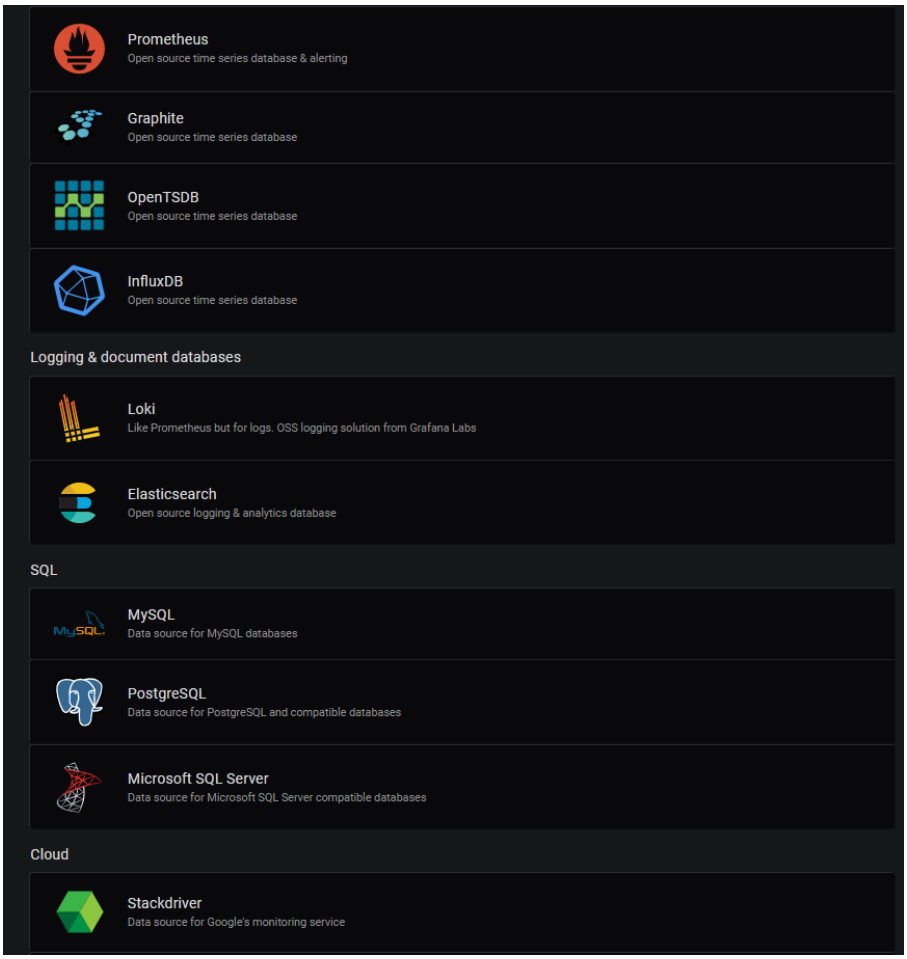

Figure 9. Some of the databases supported by Grafana [13]

In terms of the variety of visualizations, both sides offer a relatively similar selection:

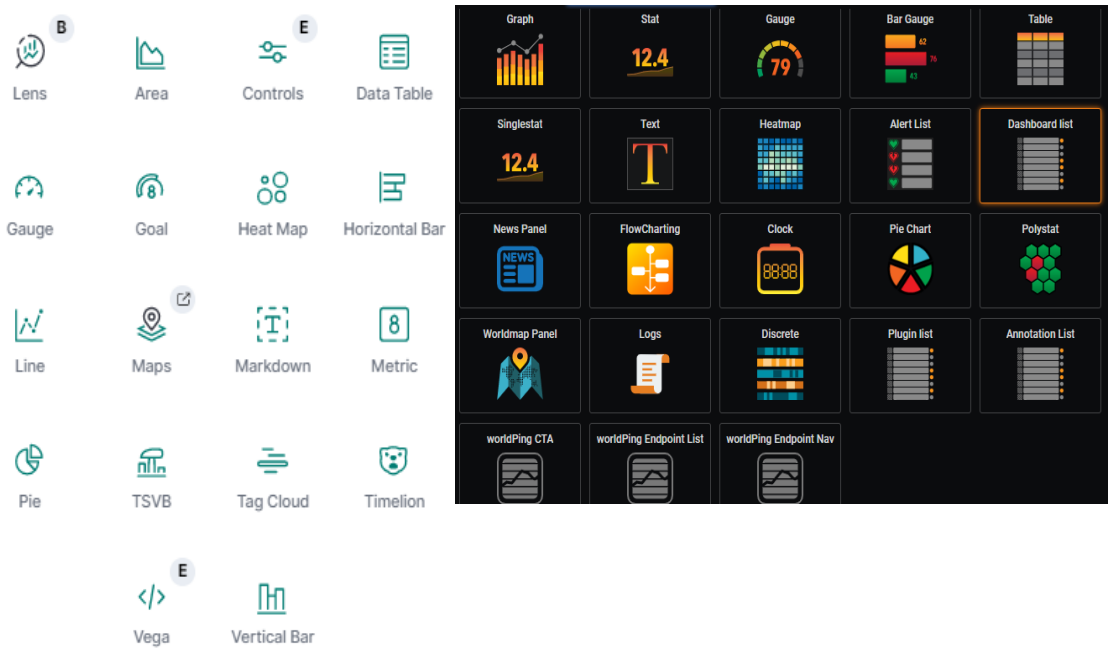

Figure 10. Comparison between Grafana and Kibana visualization choices [13] [14]

From the figure above, most basic graphs are covered in both Kibana and Grafana, as well as some more complex ones. There are also external plugins that are developed by the community available for users to install, should they need features that are not yet officially implemented for Grafana [15] or Kibana [16].

Since the purpose of this research is to determine if it is possible to replace the d3.js library with another visualization software, a comparison needed made in order to choose the better one out of the two. The table below shows how the comparison process was done:

Table 1. Summary of the comparison between Grafana and Kibana

	Grafana	Kibana
Database	Most popular databases are available	Only Elasticsearch
Visualization	Similar choices, with external plugins for additional features	
Installation	Graphical installation available	Only command line

There are other minor differences between the two as well; nevertheless, on the basis that Grafana accepts almost every popular database (even Elasticsearch itself), it has been chosen as the visualization software to test the replicability of VISDOM-project's visualizations.

3. INVESTIGATION ON THE SUITABILITY OF GRAFANA

This chapter focuses on discussing the whole process of using Grafana to create similar visualizations to that of the VISDOM-project. With some available materials as starters, the goal is now to create visualizations that are similar to that of VISDOM-project.

3.1 Available materials

Following Figure 7, the first step in doing data visualization is to get data from a source (be it a sensor sending data or software logs, etc.). For the sake of simplicity, the data of this task has already been given in the form of JSON files. It is also assumed that the data has already been neatly pre-processed; that means there will be no corrupted entries, and it is already capable of being put into use.


```
"_id": "5dfce7e0424731001109d570",
"time": "2019-05-16T13:31:32.212Z",
"creator": "unknown",
"type": "state change",
"statechange": {
    "from": "",
    "to": "Doing next"
},
"updated": "2019-12-20T15:25:20.212Z",
"__v": 0,
"related_events": [],
"related_constructs": ["5dfce7de424731001109d268"],
"isStatechange": true,
"duration": 0,
"origin_id": [{
    "source_id": "13157",
    "source": "gitlab",
    "context": "repolainen",
    "_id": "5dfce7e0424731001109d571"
}]
```

```
"_id": "5dfce7de424731001109d2f0",
"time": "2019-11-19T07:37:33.173Z",
"creator": "heikkin2",
"type": "comment",
"data": {
    "message": "changed the description"
},
"updated": "2019-12-20T15:25:18.705Z",
"__v": 0,
"related_events": [],
"related_constructs": ["5dfce7de424731001109d23e"],
"isStatechange": false,
"duration": 0,
"origin_id": [{
    "source_id": "78940",
    "source": "gitlab",
    "context": "repolainen",
    "_id": "5dfce7de424731001109d2f1"
}]
```


Figure 11. Example data from events1.json (left) and events2.json (right)

It should be noted beforehand that the data used in this research are but a few samples of the real data, for the sake of confidentiality. In total, there are 2 available JSON files: events1.json and events2.json, with each file's general structure listed in Figure 11. These entries represent changes in a certain working repository of Tampere Universities, and each key-value pair gives some more details about that change, i.e. what has happened, when it happened, etc. A brief observation revealed that the two JSON data

are strikingly similar in terms of structure, thus it was decided that the data will be joined together to make a new file called `events.json`.

One of the features currently implemented by VISDOM-project is to count the total amount of commits of each member during a certain time frame using a graph, or illustrate a comprehensive timeline for each issue available. The visualizations from VISDOM-project are given beforehand to be used as targets:

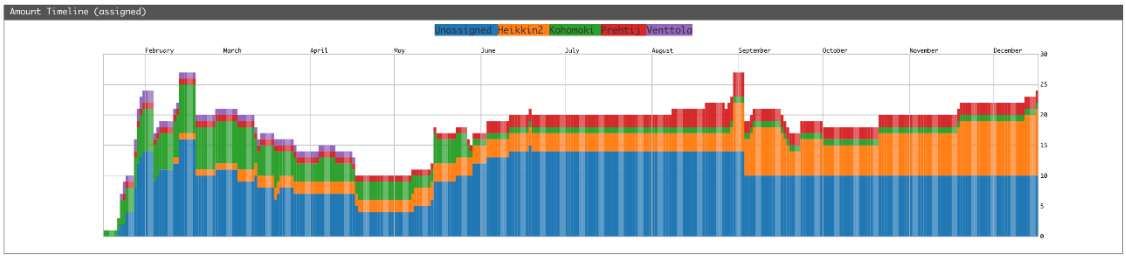

Figure 12. First visualization template

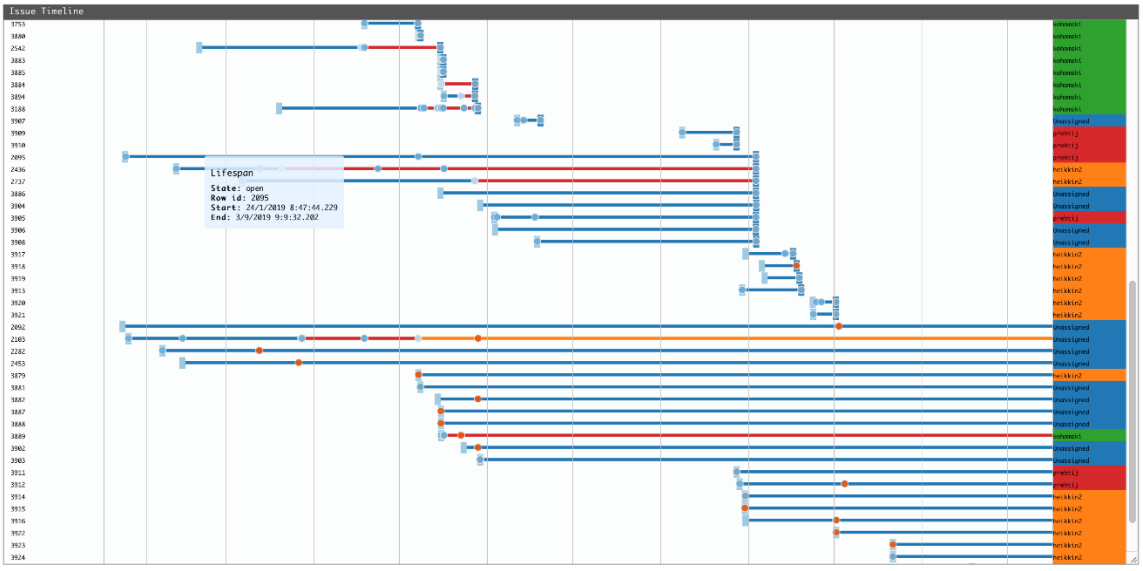

Figure 13. Second visualization template

Figure 12 illustrates the amount of commits a certain person (represented by their username) made during a constant time frame; here, the person is determined by the ” `creator` field in both files, whereas the time is determined by `time` field.

In Figure 13, all entries with the same `source_id` field will be grouped into a single lifetime row, and each JSON entry will represent a certain stage of the issue, which lasts from the `time` field to the `updated` field. The names on the right shows the last person responsible for that issue.

3.2 Conversion of data to be used in Grafana

With the start (JSON data) and the goal (above-mentioned visualizations) already defined, it means that the first step described in Figure 7 is done. It is now time to move on to the second phase of the process by putting the data into a database accepted by Grafana.

It has been discussed in Chapter 2.3 that Grafana cannot process JSON-data as-is; instead, it has to be put into a form of database that Grafana accepts (a brief list of that can be found in Figure 9). For this research, the chosen database is PostgreSQL version 12 [17], due to the fact that it is a popular database, therefore any issues arising during the implementation should have a high chance of being solved already. Additionally, the writer has had some prior experience with PostgreSQL as well, making it a good fit for this project.

PostgreSQL supports loading data into its servers by either “restoring” from a pre-compiled database, or using queries, and the latter has been chosen due to its simplicity, and also the fact that there are no ready-to-restore databases for this data yet. This in turn brings up the problem of converting the JSON data into SQL queries suitable for PostgreSQL. A number of ideas was proposed for quick conversion from JSON into SQL, namely:

- Using online websites to automatically convert the given JSON to SQL queries
- Writing a script for file reading and writing, using any programming language

Both of these solutions will be attempted, and the results will be shown in the following subsections.

3.2.1 Using online converters

The first solution was tested by going into an online converter website called sqlizer.io. As expected, the process went very quickly; one simply puts the file in and checks the appropriate options before clicking the green `Convert My File` button, and after a few moments an SQL query will be generated.

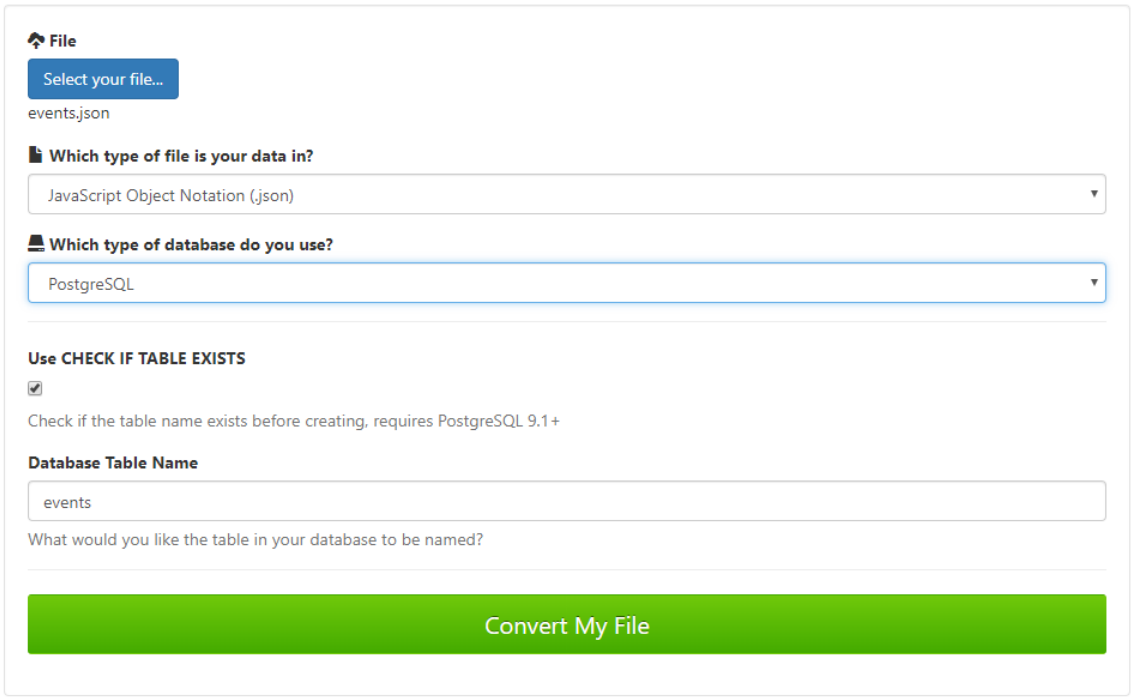

Figure 14. Example JSON to SQL conversion website UI [18]

The generated SQL query is then put into the Query Editor of the pgAdmin version 4.18 [19] application, which is the GUI version of PostgreSQL (alternatively, it is also possible to load the SQL query using the command line version of PostgreSQL, but it requires a bit more effort). The Query Editor can be easily accessed by clicking the leftmost button (marked in a red circle in Figure 15) on the top left `Browser` panel after pointing to the right database:

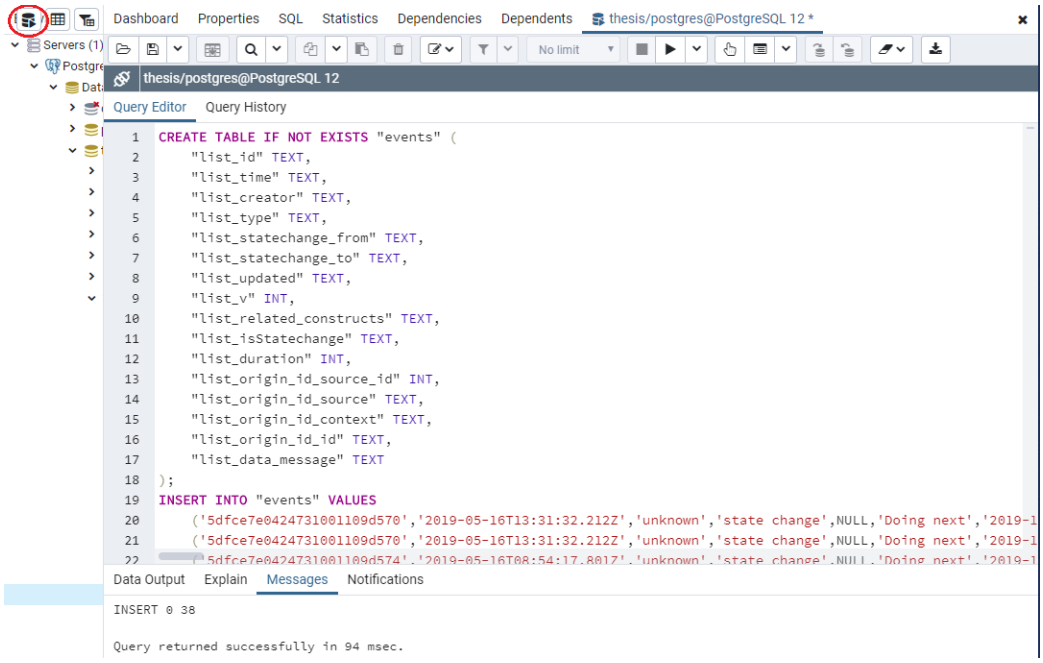

Figure 15. Snippet of the query and the result of running it [19]

As seen from the above image, the query was executed without any errors, and one can check if the data really exists in the database by typing a simple query:

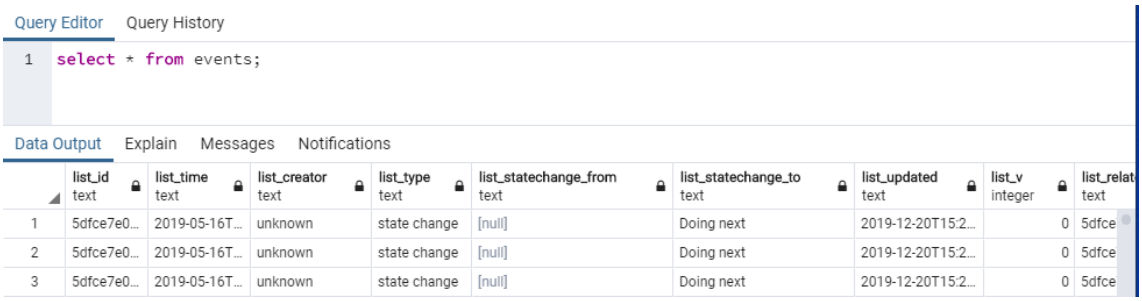

Figure 16. One-line verification query [19]

The query should return the whole `events` table if it exists [20], and in Figure 16 it can be seen that some data was shown in the output, meaning data has been added to the `events` table. However, the data does not seem to be acting as normally expected, particularly at the later columns of the table:

list_origin_id_source_id integer	list_origin_id_source text	list_origin_id_context text	list_origin_id_id text	list_data_message text
13157	[null]	[null]	[null]	[null]
[null]	gitlab	repolainen	5dfce7e0424731001...	[null]
13153	[null]	[null]	[null]	[null]
[null]	gitlab	repolainen	5dfce7e0424731001...	[null]
13155	[null]	[null]	[null]	[null]
[null]	gitlab	repolainen	5dfce7e0424731001...	[null]
13118	[null]	[null]	[null]	[null]
[null]	gitlab	repolainen	5dfce7e0424731001...	[null]
13158	[null]	[null]	[null]	[null]
[null]	gitlab	repolainen	5dfce7e0424731001...	[null]
13149	[null]	[null]	[null]	[null]
[null]	gitlab	repolainen	5dfce7e0424731001...	[null]
13122	[null]	[null]	[null]	[null]
[null]	gitlab	repolainen	5dfce7e0424731001...	[null]

Figure 17. Irregular patterns on the "events" table [19]

For example, comparing the first row to the data of Figure 11, data from the `list_origin_id_source`, `list_origin_id_context` and `list_origin_id_id` is missing and replaced by `null` values. The query generated by sqlizer.io also reflects the same phenomena. A closer look at the table yields the result that every 2 rows of the table represent a single entry in the `events.json` file, but for some reason was split into 2 rows and the last columns being separated as seen above.

Figure 18. Repetition of table rows in events.json SQL query [19]

The figure is a bit small, but for the sake of clarity, most table columns have to be included. Figure 18 shows that the two `id`s of the data rows are exactly the same with each other, which is not normally possible since `id`s are supposed to be unique strings, therefore it can be concluded that there has been a repetition in the rows of the table. This issue shows that it is not possible to use the automatically converted queries as-is in the Query Editor; there needs to be some modifications to the query itself. However, that would defeat the purpose of convenience in using ready-made queries and modifying thousands of lines of queries is a very tedious task to say the least. Therefore, the idea of using converters to quickly generate queries was scrapped, thus leaving the idea of making a conversion script the only option remaining.

3.2.2 Creation of data conversion script using Python

There are certain advantages of creating a script to convert the JSON data. First of all, one can give proper names to the columns of the table; it can be seen that the columns from the query generated by the online converter were not very appropriately named as they all begin with `list_`, making it rather unintuitive when the need to reference them comes. Secondly, the type of columns can be correctly defined; the converter can only distinguish between `TEXT` and `INT` type and nothing else, and as seen from the example entries, there are text arrays and JSON arrays that needs to be recognized in order to generate the correct table. Last but not least, converters cannot deal with irregularities in the data. For example, given a `Boolean` column in the table, if it's `True`, then from that spawns a list of follow-up values, but if it's `False`, then that list will not occur:

```
    "_id": "5dfce7e14247310011O9d758",
    "time": "2019-11-18T11:40:25.926Z",
    "creator": "heikkin2",
    "type": "updated",
    "updated": "2019-12-20T15:25:21.211Z",
    "__v": 0,
    "related_events": [],
    "related_constructs": ["5dfce7de4247310011O9d242"],
    "isStatechange": false,
    "duration": 0,
    "origin_id": [{
        "source_id": "3925",
        "source": "gitlab",
        "context": "repolainen",
        "_id": "5dfce7e14247310011O9d759"
    }]
}, {
    "_id": "5dfce7e14247310011O9d75a",
    "time": "2019-01-17T13:33:06.591Z",
    "creator": "venttola",
    "type": "opened",
    "statechange": {
        "from": "",
        "to": "open"
    },
    "updated": "2019-12-20T15:25:21.204Z",
    "__v": 0,
    "related_events": [],
    "related_constructs": ["5dfce7de4247310011O9d2f2"],
    "isStatechange": true,
    "duration": 0,
    "origin_id": [{
        "source_id": "-162",
        "source": "gitlab",
        "context": "repolainen",
        "_id": "5dfce7e14247310011O9d75b"
    }]
```


Figure 19. An irregularity in the events.json file

From the figure above, if `isStatechange` is true (as with the lower entry), then there exists a new key called `statechange`, but if the `Boolean` is `False`, then the `statechange` key does not exist in the data. The converter simply cannot detect such patterns, making it hard to debug the errors when the query is used.

For the conversion script, any programming language that is capable of file handling operations like read/write text lines is sufficient, and Python version 3.7.6 [21] was

chosen due to the writer having prior experience in using the language. The created script is called `events_to_postgresql.py`, and the beginning of the file is as follows:

```
1. import json
2.
3. with open("events.json", 'r') as jsonfile:
4.     jsondata = jsonfile.read()
5. json_list = json.loads(jsondata)
6.
7. col_names = [
8.         "_id",
9.         "time",
10.        "creator",
11.        "type",
12.        "statechange_from",
13.        "statechange_to",
14.        "message",
15.        "last_updated",
16.        "__v",
17.        "related_events",
18.        "related_constructs",
19.        "isStatechange",
20.        "duration",
21.        "origin_source_id",
22.        "origin_source",
23.        "origin_context",
24.        "origin_id"
25.]
```

Listing 1.Initialization of the conversion script

The first line defines the only Python library needed for conversion, the `json` library [22]; it is used to process the JSON data. Lines 3 and 4 utilizes the file handling features of Python, particularly `open` and `read`, to get all data from the `events.json` file into a string called `jsonfile` [23]. Afterwards, the long string is processed by the `json` library using the loads function, generating a `list` of `dict` objects (consisting of key-value pairs)

called `json_list` [23]. The JSON file is no longer needed, as we now only need the `json_list` variable. To start making the query, first it is necessary to prepare a `list` of desired table column headings to make querying the data easier and more convenient. The `list` was stored in the variable called `col_names`.

With all the needed initializations done, it is now possible to move to generating the query into a text file.

```
26.with open("events.sql", 'w') as sqlfile:
27.    sqlfile.write('DROP TABLE IF EXISTS "events";\n')
28.    sqlfile.write('CREATE TABLE "events" (\n')
29.    for cols in col_names:
30.        line = ""
31.        if cols in ["duration", "__v", "origin_source_id"]:
32.            if cols == "origin_source_id":
33.                line = "\t\"" + cols + "\" INT [],\n"
34.            else:
35.                line = "\t\"" + cols + "\" INT,\n"
36.        elif cols in ["related_events", "related_constructs",
37.                      "origin_context", "origin_source", "origin_id"]:
38.            if cols == "origin_id":
39.                line = "\t\"" + cols + "\" TEXT []\n"
40.            else:
41.                line = "\t\"" + cols + "\" TEXT [],\n"
42.        elif cols in ["time", "last_updated"]:
43.            line = "\t\"" + cols + "\" TIMESTAMP,\n"
44.        elif cols == "isStatechange":
45.            line = "\t\"" + cols + "\" BOOLEAN,\n"
46.        else:
47.            line = "\t\"" + cols + "\" TEXT,\n"
48.        sqlfile.write(line)
49.    sqlfile.write(");\n")
50.    sqlfile.write("INSERT INTO \"events\" VALUES\n")
```

Listing 2. Initializing table column headings in SQL query

For this part, another feature of Python's file handling library is used, that is the `write` feature, with a small note that all prior contents (if any) from the file will be deleted [23]. If the file does not exist in the current working directory, then it will be created instead. The file `events.sql` is now opened and ready for writing, with line 27 being the first query it needs; the purpose of it is to delete any table whose name is the same as the one which is going to be created, if such a table exists [20]. This ensures that the table we are creating will only contain the data we specify below. After dropping any existing table, the next step is to create a blank table with the same name, whose columns will be specified in the lines right below [20].

Lines 29 to 48 uses the pre-made heading list in Listing 1 to generate the queries for the table's columns. For each heading, its name will be inserted along with the type specified by various `if`-clauses. Here, the type is pre-determined by looking at the first entries in the source file, and then deciding the type for each field. Line 49 is used to signify the end of heading declaration, and information will start to be inserted after the query on line 50.

```
51.     for obj in json_list:
52.         line = "\t("
53.         elements = []
54.         elements.append(obj["_id"])
55.         elements.append(obj["time"])
56.         elements.append(obj["creator"])
57.         elements.append(obj["type"])
58.         if obj["isStatechange"] == False:
59.             elements.append("Unknown")
60.             elements.append("Unknown")
61.             try:
62.                 elements.append(obj["data"]["message"].replace("'", "'
    '"))
63.             except KeyError:
64.                 elements.append("N/A")
65.         else:
66.             elements.append(obj["statechange"]["from"])
67.             elements.append(obj["statechange"]["to"])
68.             elements.append("N/A")
```

```
69.         elements.append(obj["updated"])
70.         elements.append(obj["__v"])
71.         elements.append(obj["related_events"])
72.         elements.append(obj["related_constructs"])
73.         elements.append(obj["isStatechange"])
74.         elements.append(obj["duration"])
75.         context = []
76.         source = []
77.         source_id = []
78.         _id = []
79.         for origin in obj["origin_id"]:
80.             context.append(origin["context"])
81.             source.append(origin["source"])
82.             source_id.append(int(origin["source_id"]))
83.             _id.append(origin["_id"])
84.         elements.append(source_id)
85.         elements.append(source)
86.         elements.append(context)
87.         elements.append(_id)
```

Listing 3. Preparation for data insertion

Listing 3 represents the process of forming an SQL query for a single data entry. First of all, an empty string is initialized and ready to be inserted with values. The key point in PostgreSQL table insertion is that the format of the data must match that of the table headings; this means the first field in the data goes to the first column, and so on [20]. This can be seen from the figure below:

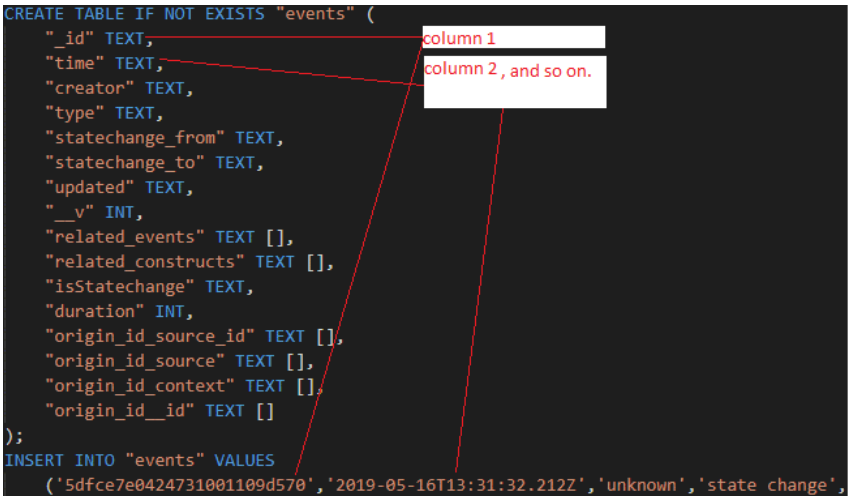

Figure 20. An illustration of how SQL table insertion works

With that in mind, the idea of Listing 3 is to go through each `dict` object in the list, which represents a JSON entry, and sequentially insert each value from the key-value pair into an intermediate list called `elements` first. This can be observed in lines 53 through 87. Although most insertions are fairly straightforward, there were some irregularities that needed to be put into concern, as discussed in this chapter (below Figure 19). The issue was addressed in lines 58 to 68, where it checks the value of the `isStatechange` field and gives appropriate actions based on whether the `Boolean` is `True` or `False`. A thing to notice in line 68 is that all single quotes (‘) in the data is replaced by double single quotes (‘’) due to PostgreSQL regulations on escape characters; so to represent a string like `student’s`, the single quote has to be escaped into `student’’s` [20]. After appending everything into the intermediate list `elements`, the next step is to join them together into a query line.

```
88.         for e in elements:
89.             if type(e) is int:
90.                 line += str(e)
91.                 line += ","
92.             elif type(e) is list:
93.                 if not e:
94.                     line += "NULL,"
95.                 else:
96.                     line += "ARRAY "
97.                     line += str(e)
98.                     line += ","
```

```
99.             else:
100.                    line += "'"
101.                    line += str(e)
102.                    line += "',"
103.            line = line[:-1]
104.            if obj == json_list[-1]:
105.                line += ");\n"
106.            else:
107.                line += "),\n"
108.            sqlfile.write(line)
```

Listing 4. Joining value fields into SQL table data format

Listing 4 is used to generate an SQL line of data using the generated intermediate list. Sequentially, for each element in the intermediate list, it will be encapsulated in single quotes if it is a `string` or `Boolean`. If the data is an integer or `NULL` (a special PostgreSQL type of data), then no quotation marks is needed. Otherwise, if the entry is a `list`, then it has to begin with `ARRAY`, followed by square brackets surrounding the appropriate format discussed above in this paragraph. During the process, each column's data will also be separated by commas. Lines 104 to 107 is used to check if the current data is the last entry of the source file or not, so that the last comma used to separate different JSON data is replaced by a semicolon, which signifies the end of the query. Finally, after an SQL line is successfully formed using one JSON entry, it is written in the SQL file and the whole process begins anew with another entry. An example line looks like below:

```
('5dfce7e0424731001109d570','2019-05-16T13:31:32.212Z','unknown','state change','','Doing next','2019-12-20T15:25:20.212Z',0,NULL,
ARRAY ['5dfce7de424731001109d268'],'True',0,ARRAY ['13157'],ARRAY ['gitlab'],ARRAY ['repolainen'],ARRAY ['5dfce7e0424731001109d571']),
```

Figure 21. Ideal SQL line of data

The completed query is then put into the Query Editor of pgAdmin and the results can be observed below:

Figure 22. Result of inserting data using self-made query [19]

Querying the created table generates the following output:

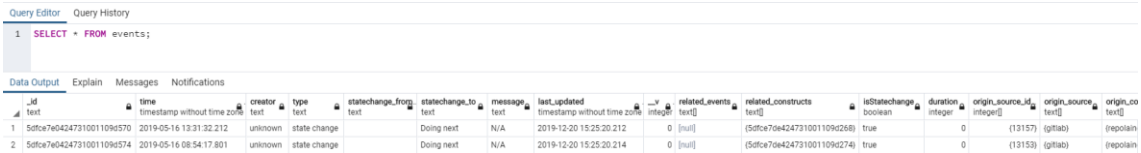

Figure 23. Generated table result [19]

From the above figures, it is safe to conclude that option two of the data conversion has been successfully implemented, and thus ready to be used in the visualizations.

3.3 Turning data into visualizations

As discussed in chapter 2.3, Grafana will be used as the visualization tool for this task (the installation of Grafana is fairly simple and will not be covered here), and to begin, the database containing the necessary data needs to be added to Grafana as a data source. This can be done from the front page of Grafana by clicking on the `Create a data source` button (indicated by the red circle in Figure 24), which will lead to an interface like in Figure 9, and then choose `PostgreSQL` from the list:

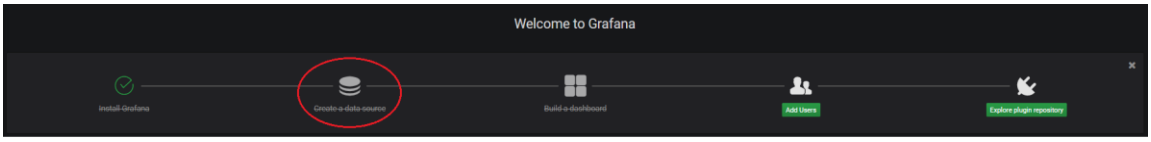

Figure 24. Start Grafana and import an existing database [13]

Choosing `PostgreSQL` will lead to a dialogue box like Figure 25, and filling in the correct information will allow Grafana to connect to the database, indicated by the green `Database Connection OK` box. Here, “correct information” refers to the machine’s current settings for PostgreSQL, and it is unique for each machine. However, please

note that these configurations are only for testing purposes; it is not applicable in a production environment, where many security and networking problems need to be taken into consideration.

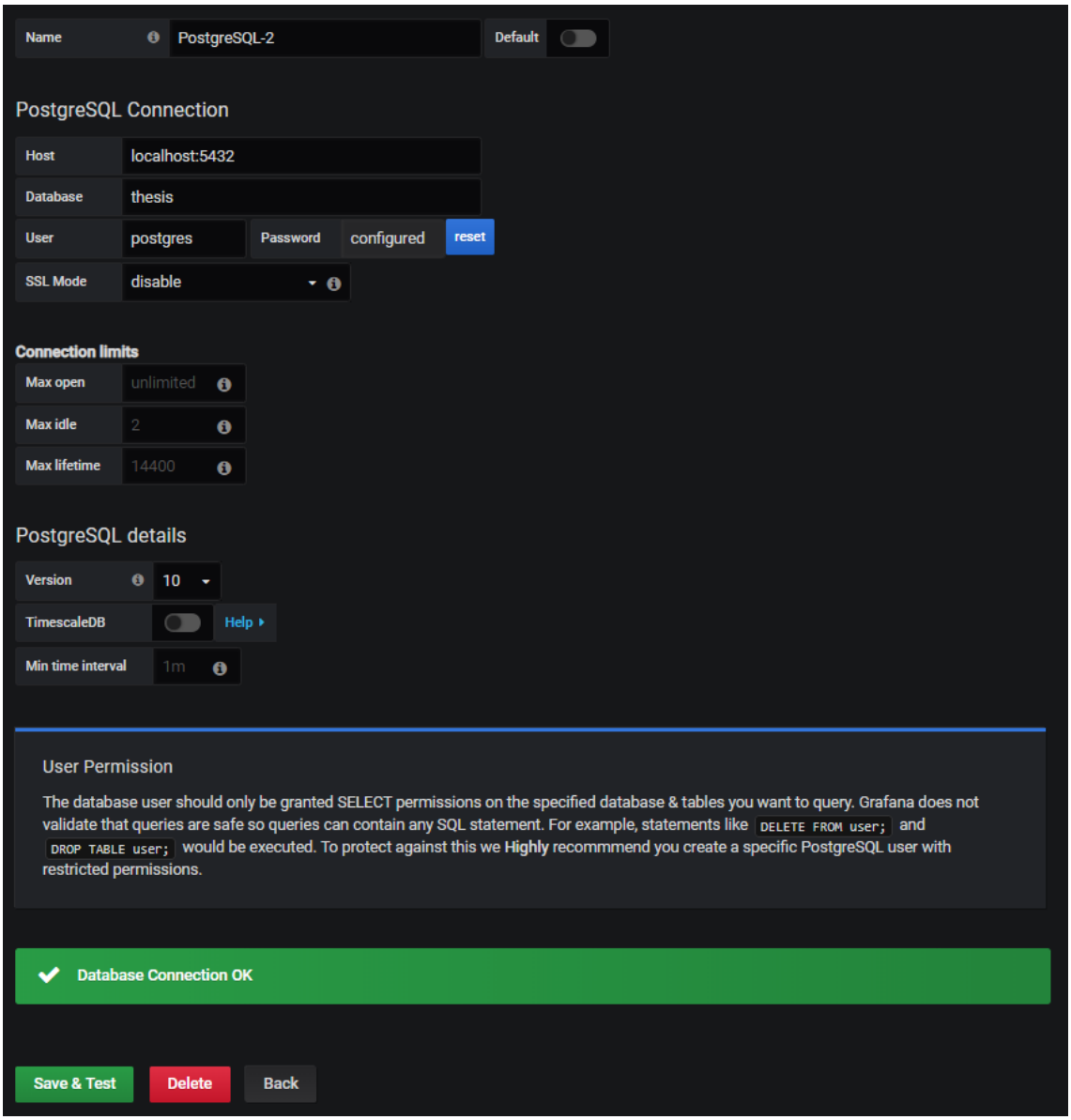

Figure 25. Adding an example PostgreSQL database [13]

After connecting the proper database, it is possible to proceed with replicating the figures from VISDOM-project.

3.3.1 First visualization template

The first graph to be implemented is the graph from Figure 12; first of all, a blank dashboard needs to be initialized by hovering on the plus sign from the left-most panel of Grafana, then choose `Dashboard`, as shown in Figure 26:

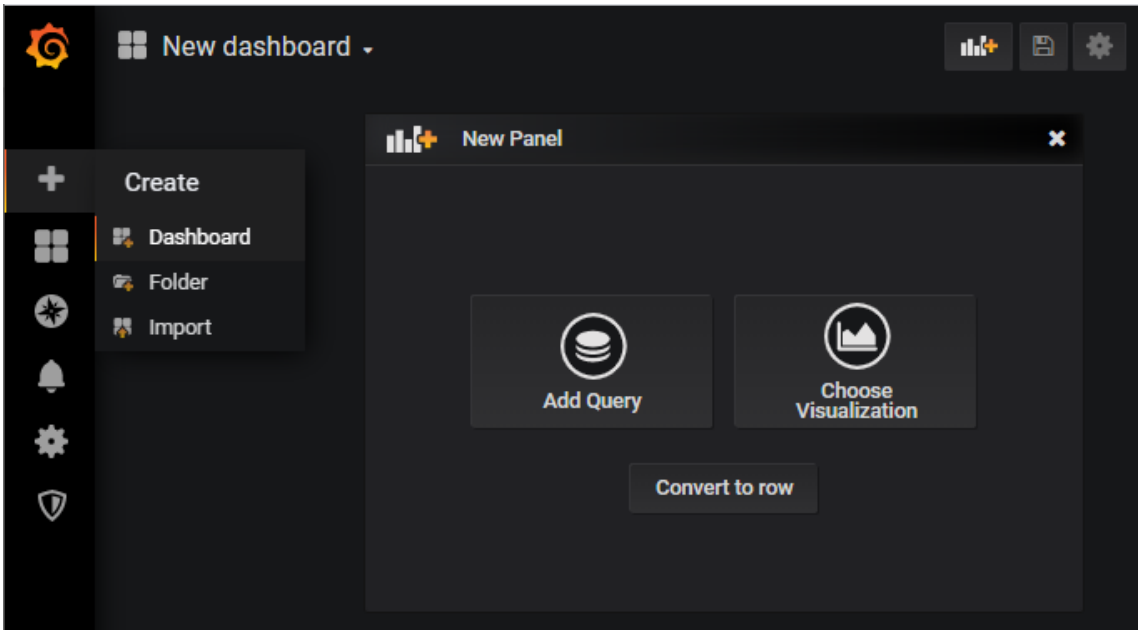

Figure 26. Create a blank dashboard in Grafana [13]

It was not revealed beforehand what type of visualization the first graph represents, but from the writer's experience, it can be most likely either a bar graph, a staircase line graph or a histogram. However, since there are no signs of the differentiation of bars in Figure 12, it is safe to narrow the options down to only the latter two. At this point, no further assumptions can be made, so the writer has decided to choose the visualization type to be a staircase line graph.

Next, it is possible to choose either of the two options on the `New Panel` box, but `Choose Visualization` is the recommended option. This allows users to specify the type of visualization before giving the query. Since the goal is a staircase line graph, the correct option to choose is the `Graph` type, and the options as specified below:

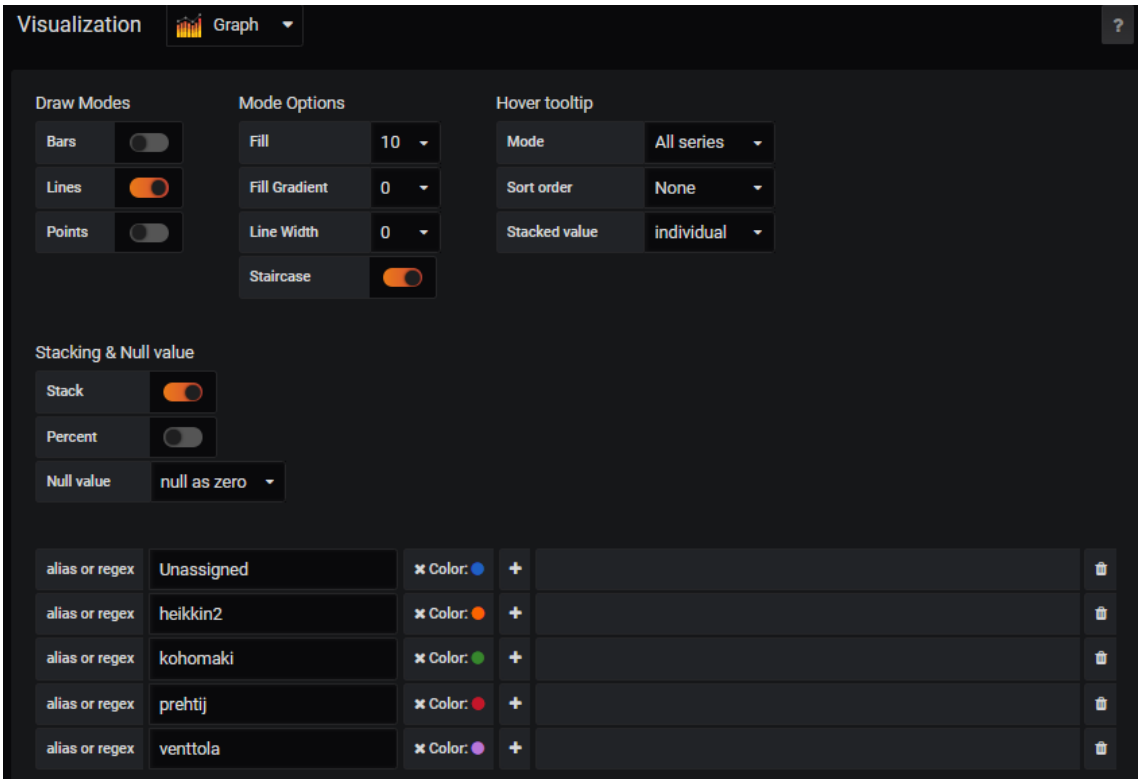

Figure 27. Options for the staircase line graph [13]

Aside from the obvious `Lines` and `Staircase` options, the `Stack` option allows for the bars to stack on top of each other instead of overlapping, and the `Fill` option must be set to maximum so that the bars will not become too transparent and blend in with the background. There has been no data given yet, but the color has been pre-determined to match with that of the original visualization. With the settings completed, it is necessary to adjust the time frame of the visualization to match with that of the data. This can be done by first finding out the maximum and minimum time from the `time` column of the data, using a simple query:

```
1  SELECT max(time), min(time) FROM events;
```

Data Output | Explain | Messages | Notifications

	max timestamp without time zone	min timestamp without time zone
1	2019-12-16 09:08:23.614	2019-01-17 13:33:06.591

Figure 28. Query for finding the time frame of the data [19]

From Figure 28, it can be seen that the earliest time is 17th of January 2019 (only the date is taken into consideration for this task), and the latest is 16th of December 2019. Therefore, it is reasonable to adjust the time frame of the visualization by clicking the clock icon on the top-right of Grafana, then adjust the fields in the `Absolute time frame` section as below:

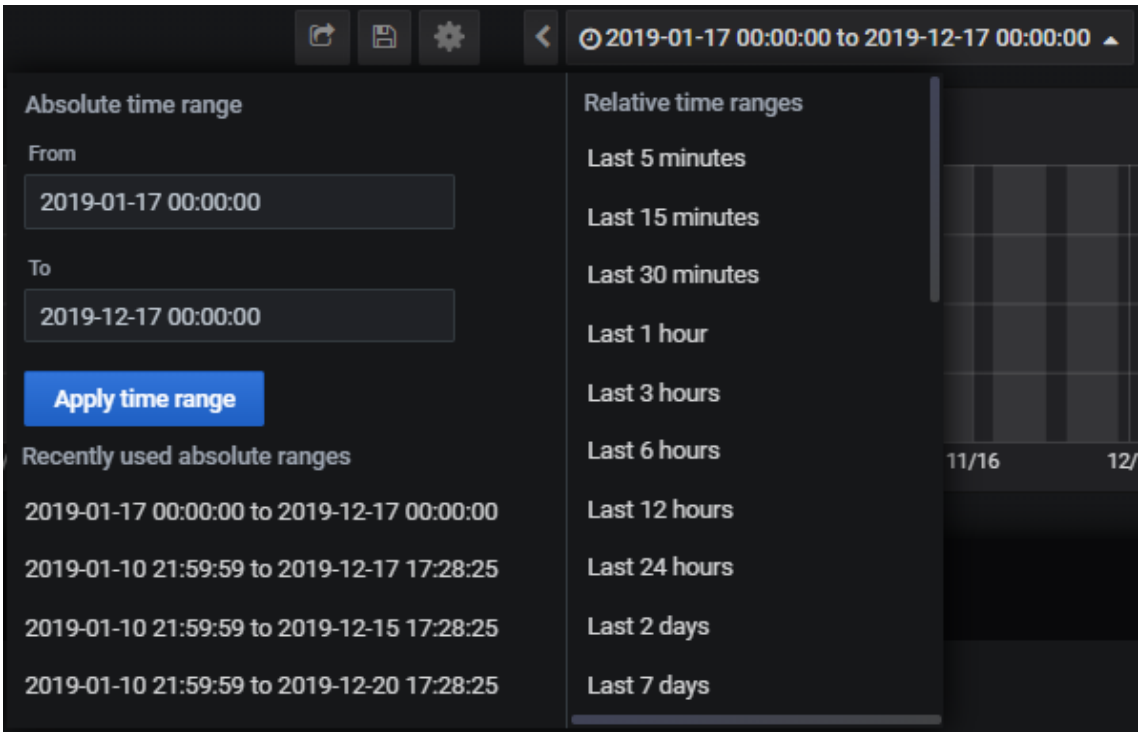

Figure 29. Adjusting the time frame to match the data [13]

The time frame has now been set from the 17th of January to the 17th of December 2019, which is guaranteed to capture all the data. Another time-related option that should be set is the `Threshold` option, which helps to distinguish between weekdays and weekends, or whatever threshold the user wants. Here, the writer wants the former purpose, so the threshold has been set from 0am Monday to 0am Saturday every week:

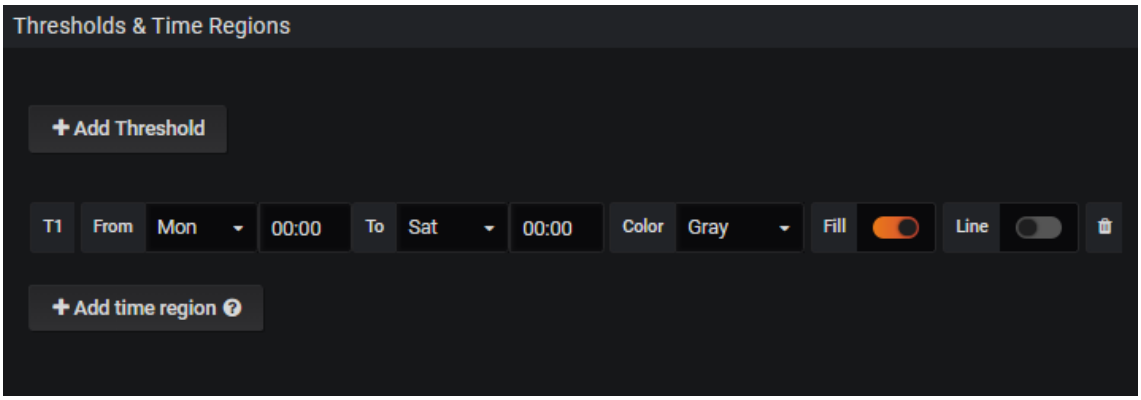

Figure 30. Weekday and weekend time thresholding [13]

All in all, the above options should generate a blank graph like below, and all that's left is to get the data in using queries:

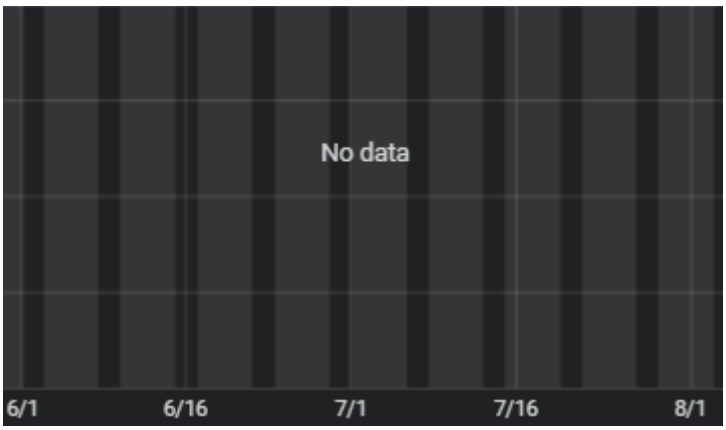

Figure 31. A snippet of the blank graph from Grafana [13]

To get the needed data for visualization, a query is formed like below:

```
1. SELECT
2.   $__timeGroupAlias("time",3d,0),
3.   count(creator) AS "Unassigned"
4. FROM events
5. WHERE
6.   $__timeFilter("time") AND
7.   creator = 'unknown'
8. GROUP BY 1
9. ORDER BY 1
```

Listing 5. Grafana query for the visualization data

The query above returns a table containing two columns, the first one being the timestamps that are grouped in three-day intervals, and the second one consisting of the total number of times that `Unknown` appeared in the `creator` field during said intervals. Some of the results can be seen below:

Data Output | Explain | Messages | Notifications

	time double precision	Unassigned bigint
1	1548201600	8
2	1548720000	4
3	1548979200	4
4	1549238400	14
5	1549756800	32
6	1550016000	8
7	1550534400	6

Figure 32. The first seven results of the Grafana query in PostgreSQL [19]

The query is Grafana-specific, meaning it is made using the format that Grafana desired, and it will not work in a SQL environment. There is actually an equivalent SQL query, and it can be found by clicking on the `Generated SQL` button below the query:

```
1. SELECT
2.   floor(extract(epoch from "time")/259200)*259200 AS "time",
3.   count(creator) AS "Unassigned"
4. FROM events
5. WHERE
6.   "time" BETWEEN '2019-01-16T22:00:00Z' AND '2019-12-16T22:00:00Z' AND
7.   creator = 'unknown'
8. GROUP BY 1
9. ORDER BY 1
```

Listing 6. Grafana query converted to PostgreSQL structure

The query, combined with the options set from Figure 27, Figure 29 and Figure 30 will produce the graph as seen below:

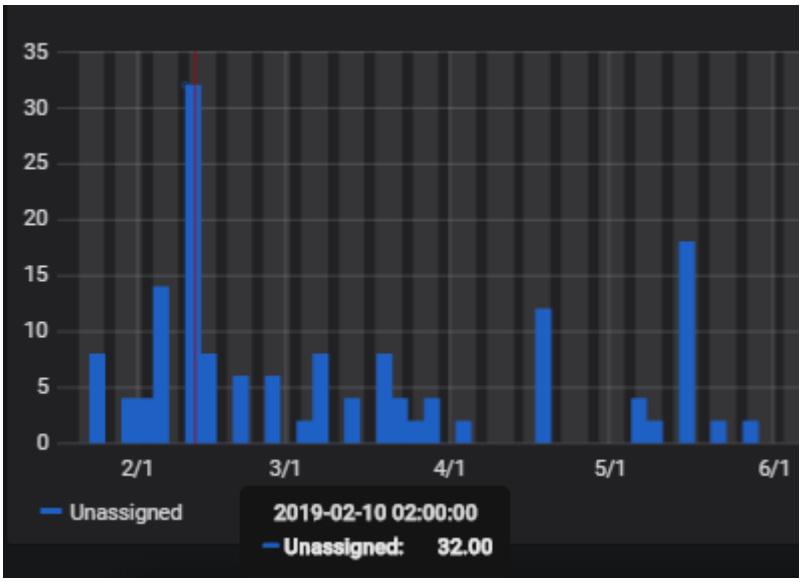

Figure 33. Resulting visualization using a single value of the "creator" field [13]

To complete the visualization, similar queries need to be called for each field of the table we want to display. Attempts were made towards designing a query that returns the counts of all distinct values during an interval without the need to repeat the query for all available values, but Grafana's usage of the `GROUP BY` clause severely limits the possibilities. In particular, since this is a time series graph, the data must be grouped by some kind of time interval or timestamp, thus disallowing the choice of grouping by the creators' names, since only one criterium can be used in the clause.

In this case, there are a total of 5 different names, as shown by the query:

```
1  SELECT distinct(creator) from events;
```

Data Output | Explain | Messages | Notifications

	creator text
1	heikkin2
2	venttola
3	prehtij
4	unknown
5	kohomaki

Figure 34. Five distinct values in the "creator" field [19]

This means that there should be 4 more queries to be filled, and they are all similar to the initial one, albeit with a slight change in labeling and value names. To be more specific, the label names can be however one feels reasonable, and the `creator` field in the `WHERE` clause should match one of the names in Figure 34. With all 5 of the queries up and running, the result should look like below:

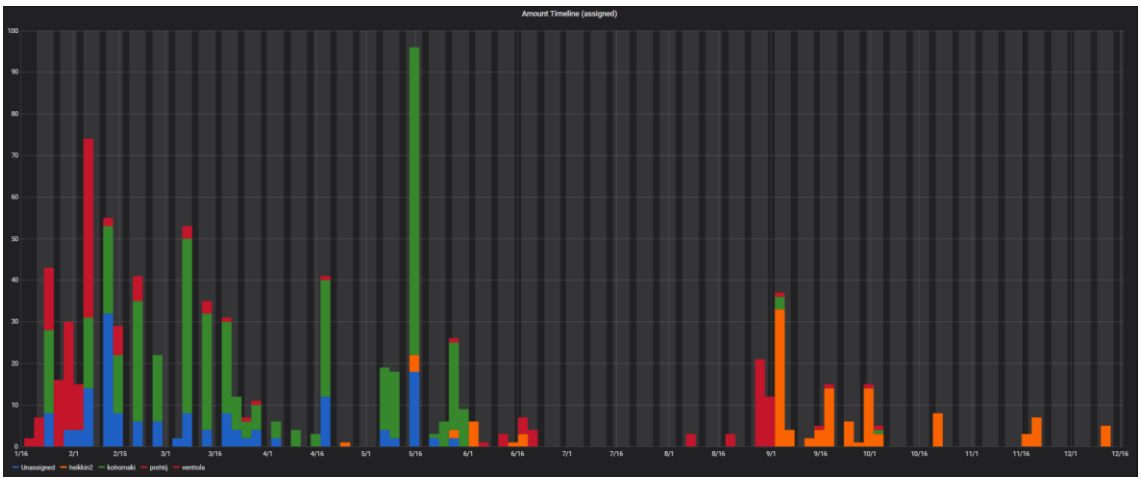

Figure 35. Completed staircase line graph [13]

Compared to Figure 12, Figure 35 offers the same structure of visualization, only slightly differing in terms of background color (white versus black), and due to the fact that data given for this experiment is not the same as the example (because of certain confidentiality reasons), the two visualizations cannot exactly match.

With the first visualization successfully replicated, the only task remaining is to make a similar representation of the second visualization.

3.3.2 Second visualization template

From the list of available types of visualizations for Grafana (see Figure 9), it can be said that there are no immediate replicas to a graph like in Figure 13. However, the external plugins section offered a potential candidate in the form of a Statusmap:

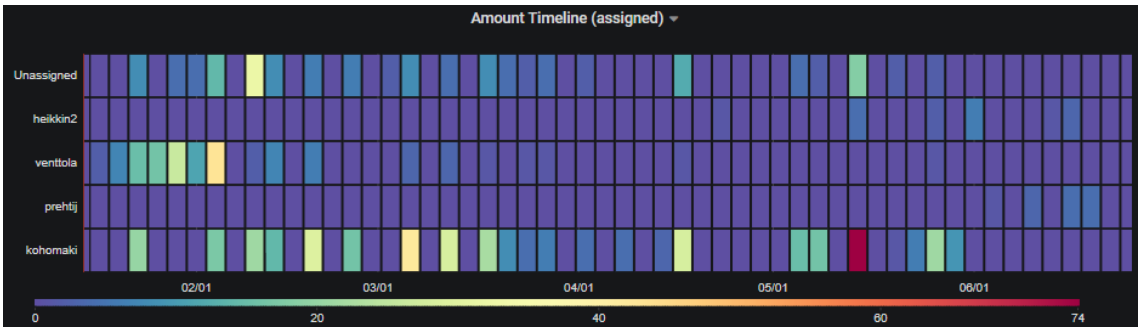

Figure 36. Example Statusmap using the former visualization's data [13]

From the above figure, the steps towards replication can be imagined as follows:

- Get all issues that have the same `origin_source_id`
- Enumerate the list of creator names from 1 to 5
- The responsibility interval of each creator for the issue is from the `time` value to the `last_updated` value
- Perform similarly to the staircase line graph, with each time interval having a value from 1 to 5

That being said, this is only a proposal towards solving this replication problem using Grafana. In doing these steps, a bit of tinkering with PostgreSQL functions is required, which the writer is yet capable enough to perform. Therefore, for now, no further steps can be taken, and the replicability of the second graph using Grafana remains in question.

4. RESULTS AND ANALYSIS

The replication process in chapter 3 reveals that it is only partially possible to use Grafana in VISDOM-project type of visualizations.

While the first graph (the staircase line graph) can be successfully and easily implemented, there has yet to be any concrete solutions for the latter visualization. What's more, the process of adding data to the first graph was found to be quite manual and laborious, with each distinct value requiring its own query. This is entirely doable for, say, tens, or even hundreds of entries, but with the context being the repositories of a whole university containing possibly thousands project members, the task can get tedious very quickly. Compared to the current d3.js implementation [ref], Grafana offers already the visualization views, but the data insertion process is mostly manual, whereas the opposite is true for the current VISDOM-project version.

In general, if there are only a few instances that need to be visualized, then it might be a good idea to use Grafana. However, given the context mentioned above, as well as the fact that it is not currently possible to set up the lifetime graph in Grafana, it is safest to still stick to what d3.js has to offer, as Grafana ultimately seems to be unsuitable for VISDOM-project type of visualizations.

5. CONCLUSIONS

Data visualization has evolved throughout history to become one of the most important facets of modern technology. Its applications can range from the classic hardware data logging to the recently emerging fields such as data science or data analysis. In that field, a new organization called VISDOM-project is aiming to make innovations based on existing visualization platforms. Having mainly relied on d3.js as the primary visualization library, VISDOM-project is looking for better alternatives to widen its selection of tools. For this purpose, the writer is tasked to judge whether it is feasible to use popular software such as Grafana or Kibana to perform similar visualizations as the current ones.

After a period of exhaustive search and comparison in chapter 2, Grafana is determined to be the best candidate for the task over Kibana for several criteria. Grafana's usage is then put into question in chapter 3 by attempting to replicate two graphs that are available from VISDOM-project. As a result, one of the graphs have been successfully replicated, while the other's replicability is undetermined due to the lack of a matching graph format from Grafana, which will undoubtedly render the task excessively difficult for the writer.

As for the final verdict to the question of suitability of Grafana and Kibana for VISDOM-project's visualizations, the answer is that ultimately speaking, it is better to stick to the current d3.js library implementation for various reasons, all of which have been discussed in chapter 4. Hopefully, in the future, there will be a visualization tool that can perform similar visualizations with less effort yet more convenience, efficiently replacing the d3.js library and making it easier for VISDOM-project to bring its innovations to real life.

REFERENCES

[1] V. Authors, "VISDOM," [Online]. Available: https://visdom-project.github.io/website/. [Accessed 30 March 2020].

[2] A.-L. Mattila, H. Terho, A. Luoto, H. Fooy and K. Systä, "VISDOM deliverable D2.1.2. Github," Tampere University, 17 December 2019. [Online]. Available: https://github.com/systa/VISDOM-gitlab-collector. [Accessed 18 April 2020].

[3] C.-h. Chen, W. K. Härdle and A. Unwin, Handbook of Data Visualization, Springer Science & Business Media, 2007.

[4] S. Few and P. Edge, "Data visualization: past, present, and future.," in *IBM Cognos Innovation Center*, 2007.

[5] S. A. Gore, "A Brief History of Data Visualization (and the role of libraries and librarians)," *Journal of eScience Librarianship,* vol. 7, no. 1, 2018.

[6] "Data Lakes and Analytics on AWS - Amazon Web Services," Amazon, [Online]. Available: https://d0.awsstatic.com/Solutions/Big%20Data/Big%20Data%20Redesign/QS_dynamically-optimized-graphics_big.png. [Accessed 13 April 2020].

[7] Tableau, "What is data visualization? A definition, examples and resources," Tableau, [Online]. Available: https://www.tableau.com/learn/articles/data-visualization. [Accessed 13 April 2020].

[8] K. Systä, A.-L. Mattila, H. Terho, A. Luoto and H. Fooy, "VISDOM-project Application Github," 24 January 2020. [Online]. Available: https://github.com/systa/visu/tree/dockerized/frontend. [Accessed 13 April 2020].

[9] M. Bostock, "D3.js - Data-Driven Documents," 2019. [Online]. Available: https://d3js.org/. [Accessed 13 April 2020].

[10] E. Meeks, "D3 is not a Data Visualization Library - Elijah Meeks - Medium," 11 June 2018. [Online]. Available: https://medium.com/@Elijah_Meeks/d3-is-not-a-data-visualization-library-67ba549e8520.

[11] "Kibana: Explore, Visualize, Discover Data | Elastic," Elasticsearch B.V., 2020. [Online]. Available: https://www.elastic.co/kibana. [Accessed 14 April 2020].

[12] "Grafana: The open observability platform | Grafana Labs," Grafana Labs, 2020. [Online]. Available: https://grafana.com/. [Accessed 14 April 2020].

[13] *Grafana Application,* Grafana Labs, 2020.

[14] *Kibana Application,* Elasticsearch B.V., 2020.

[15] "Grafana Plugins - extend and customize your Grafana.," Grafana Labs, 2020. [Online]. Available: https://grafana.com/grafana/plugins?orderBy=weight&direction=asc. [Accessed 13 April 2020].

[16] "Known Plugins | Kibana Guide [7.6] | Elastic," Elasticsearch B.V., 2020. [Online]. Available: https://www.elastic.co/guide/en/kibana/current/known-plugins.html. [Accessed 13 April 2020].

[17] The PostgreSQL Global Development Group, "PostgreSQL: The world's most advanced open source database," The PostgreSQL Global Development Group, 2020. [Online]. Available: https://www.postgresql.org/. [Accessed 14 April 2020].

[18] D4 Software Ltd, "Easily convert files into SQL databases | SQLizer," D4 Software Ltd, 2019. [Online]. Available: https://sqlizer.io/#/. [Accessed 14 April 2020].

[19] The pgAdmin Development Team, "pgAdmin 4," The pgAdmin Development Team, 2020. [Online]. Available: https://www.pgadmin.org/docs/pgadmin4/4.18/index.html. [Accessed 14 April 2020].

[20] The PostgreSQL Global Development Group, "PostgreSQL: Documentation: 12: PostgreSQL 12.2 Documentation," The PostgreSQL Global Development Group,

2020. [Online]. Available: https://www.postgresql.org/docs/12/index.html. [Accessed 14 April 2020].

[21] G. Van Rossum and F. L. Drake, *Python 3 Reference Manual,* Scotts Valley, CA: CreateSpace, 2009.

[22] "json - JSON encoder and decoder - Python 3.8.2 documentation," 10 April 2020. [Online]. Available: https://docs.python.org/3/library/json.html#json.loads.

[23] G. Van Rossum and F. L. Drake, "Python 3.7.7 documentation," Python Software Foundation, 2020. [Online]. Available: https://docs.python.org/3.7/index.html. [Accessed 14 April 2020].