Towards a Domain-Specific Language for Geospatial Data Visualization Maps with Big Data Sets

Cleverson Ledur, Dalvan Griebler, Isabel Manssour, Luiz Gustavo Fernandes
PUCRS, Faculdade de Informática, Computer Science Graduate Program,
Fax: +555133203621, Av. Ipiranga, 6681, CEP: 90619-900 – Porto Alegre – Brazil
Email:{cleverson.ledur, dalvan.griebler}@acad.pucrs.br, {isabel.manssour, luiz.fernandes}@pucrs.br

***Abstract*—Data visualization is an alternative for representing information and helping people gain faster insights. However, the programming/creating of a visualization for large data sets is still a challenging task for users with low-level of software development knowledge. Our goal is to increase the productivity of experts who are familiar with the application domain. Therefore, we proposed an external Domain-Specific Language (DSL) that allows massive input of raw data and provides a small dictionary with suitable data visualization keywords. Also, we implemented it to support efficient data filtering operations and generate HTML or Javascript output code files (using Google Maps API). To measure the potential of our DSL, we evaluated four types of geospatial data visualization maps with four different technologies. The experiment results demonstrated a productivity gain when compared to the traditional way of implementing (*e.g.*, Google Maps API, OpenLayers, and Leaflet), and efficient algorithm implementation.**

***Keywords*—Data Visualization Maps, Domain-Specific Language, Geospatial Data Visualization, Big Data Analysis, Code Generation, Raw Data Processing**

I. Introduction

Data generation is increasing exponentially in the last years. In 2002, approximately five exabytes of data were stored globally, and a volume of eighteen exabytes of new data were transferred through electronic ways [1]. In 2007, the amount of digital data produced in a year surpassed the worlds data storage capacity for the first time. The total amount of data generated in 2009 was eight hundred exabytes [2]. The International Data Corporation (IDC)[1] estimates that this volume would grow about forty-four times for 2020, which implies in forty percent rate of annual growth. All these data is being produced by many worldwide fields, for example, social networks, government data, health care, the stock market, among others [3].

Big Data analysis and data visualization can provide interesting information that can help in decision-making. When data sets are well-analyzed, they may predict tendencies for helping on future actions, and solving current problems. This is often used in areas like Biology, Health, Finance, Social Networking, among others. In favor of the data visualization, we have the human perception of images that can process in parallel many information. While reading texts or values, the brain processed it sequentially [4]. Consequently, data visualization is more productive on big data analysis because facilitates and accelerates the human insight over the data. The representation of information using graphics elements has evolved and has been used in several areas for increasing the human perception [5].

[1]http://www.idc.com/

However, big data analysis is still a challenge [6] due to the cost required in processing and manipulating by using the current tools and applications. Among the techniques used for big data analysis, we can highlight: artificial and biological neural networks; models based on the principle of the organization; methods of predictive analysis; statistics; natural language processing; data mining; optimization and data visualization [3].

When creating data visualization maps in today's scenario, the user generates maps inserting all data directly on HTML file format and programming with JavaScript. For large raw data sets, it becomes more difficult for dealing manually. Most of the tools for this purpose requires software development skills, even they are providing suitable libraries to create the visualization. Another problem is the data filtering because requires much legwork for reading and copying each register to the secondary file. Finally, the most complex is the classification. It uses the result of the filter process, and it is up to the user choose the appropriate classification algorithm, provide a custom class, or use an external tool. In addition, it is necessary to transform the classified output in the JavaScript library format.

Current map visualization scenario obliges at least to know/learn two or more web programming languages and learn how to work with a pre-processor tool or build one from the scratch. We are proposing an external DSL [7]–[9] that abstracts all these knowledge needed, where only a specification-based language with a small dictionary, simple syntax, and familiar geospatial visualizations keywords has to be learned. Moreover, it is designed to support automatically raw data manipulation, pre-processing, filtering, classification, and visualization creation. All these abstractions are not taking the user's power for fill the visualization needs. The idea is just to specify the operations while the DSL's compiler handles its implementation. Therefore, the main contributions are:

- A DSL designed for data scientists to generate geospatial data visualization maps with raw big data sets.
- An high-level abstraction able to improve the productivity compared with current technologies (Google Maps API, Leaflet, and OpenLayers).
- An efficient data processing implementation, which is generated by the DSL's compiler.

The remainder of this work is organized as follows. Sections II and III presents the background and the most important related work. Section IV details the proposed domain-specific language. The used methodology for the evaluation is described in Section V. Section VI describes the experiments and evaluates the performance and the code productivity. Finally, Section VII presents the conclusions and future works.

II. Background

Geospatial data visualization uses a special type of data that specify the location of an object or phenomena [4]. Generally, this is possible due to the information of *Latitude* e *Longitude* in each register. Examples of geospatial data are global climate modeling, environmental records, economic and social measures and indicators, customer analysis, and crime data. The strategy used to represent this kind of data is to map spatial attributes directly to the two physical screen dimensions, resulting in map visualizations.

In creating a data visualization, it is important to know about the input, structure and kind of data that we are handling. When this data come from an external source, it is necessary to perform data preparation for selecting, filtering and cleaning. After, it is possible to map data into visual representations, according to the attributes previously selected. Finally, the visualization generation can be done [4].

There are some libraries that allow users to create data visualization maps. When using these libraries, like Google Maps API, OpenLayers, and Leaflet, it is up to the user preprocessing the data in the correct format. When dealing with big data sets, users will have to work hardly and consume more time to plot a map. Usually, it will worth to create a software for automating data processing. Figure 1(a) shows the workflow of traditional libraries to generate a visualization. The dotted line around *Generation of the Visualization* and *Library Format Data* demonstrates the scope of these libraries support without the needing of extra programming by the user.

A. Google Maps API

Introduced by Google in 2005, this API revolutionized the way we use maps on the web, allowing users to drag and interact the visualization to find the expected information. Google Maps API operates using HTML, CSS, and JavaScript working together. The map tiles are pieces of images that are loaded in the background with Ajax calls and then inserted into a <div>in the HTML page. When navigating through the map, the API sends information about the new coordinates and zoom levels of the map in Ajax, which returns new images [10].

For creating data visualization maps using Google MAPs API, the user needs to have knowledge in JavaScript for creating variables, objects and use functions. Initially, this requires the inclusion of the library in the HTML file and the association of a map object to a variable and information inside, considering map details like initial position, zoom, and layer. After, for each marker can be created a limit of three lines of code to generate an object with latitude and longitude information. If a classification is desired, like to change marker colors, users may also insert in the JavaScript code a tag for an icon declaration.

B. OpenLayers

OpenLayers is an open source JavaScript library that provides features for displaying map data in web browsers. Also, it provides an API for building web-based geographic applications. Furthermore, this presents a great set of components, such as maps, layers, or controls. OpenLayers offers access to a great number of data sources using many different data formats, and implements many standards from Open Geospatial Consortium[2] [11].

Also, it requires the insertion of the library in the HTML file and a library creation before adding a new marker. The layer is an OpenLayers feature in that allows the implementation of different types of data visualization in a single data visualization. Google Maps API and Leaflet, presented in Subsection II-C, abstracts this option. Consequently, they are limiting for users to show just one type of data visualization. This possibility of layers creation impacts in the marker insertion process, because each marker may be created and associated to a layer. If a simple data visualization map with just one layer may be created, this generates more lines of code than in other layers.

C. Leaflet

Leaflet is an open-source JavaScript library for the creation of interactive maps. Leaflet works taking advantage of HTML5 and CSS3 [12], and also allows the creation of maps using geospatial data. Moreover, it provides tile layers, markers, popups, vector layers like polylines, polygons, circles, rectangles, circle markers, GeoJSON layers, image overlays, WMS layers and layer groups.

The marker and map creation in Leaflet is similar to others libraries. Initially occurs the insertion of the library into the HTML file and a set of codes is responsible for creating the map as well as the markers are inserted later. An option allows the specification of information inside markers when it is selected by a click. As in others libraries, the user must have knowledge in JavaScript programming to create a data visualization map and the replication of code for each marker inserted.

III. Related Work

In data visualization domain, some DSLs have been proposed looking for increasing the visualization creation possibilities for some domain-specific users. They have focused on some domain and does not support the creation of standard visualizations. Vivaldi [13] aims at facilitating the visualization creation and volumetric processing on heterogeneous systems. By other hand, ViSlang [14] operates in scientific visualization, where its main contribution is to provide extensibility. Diderot [15] looks for simplifying image analysis, using a language with tensors to approximate the users' domain. Finally, Shadie [16] was designed to create efficient scientific big data visualizations.

Superconductor [17] provides a set of generic big data visualization features. It is not designed for a specific visualization type. Also, it allows the user to personalize the

[2]http://www.opengeospatial.org

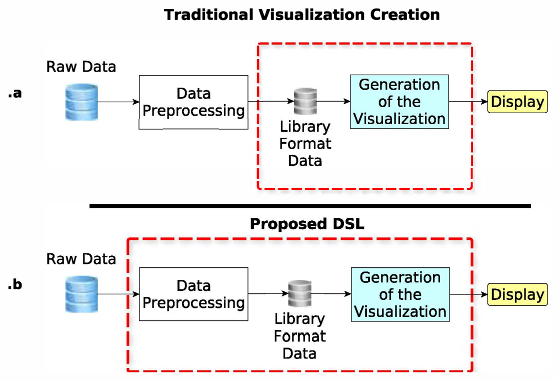

Fig. 1. Visualization creation comparison.

visualization by changing visual elements. Comparing to our work, Superconductor is more robust. However, users have to build their visualizations from the scratch interacting with low-level programming stuff, while we are abstracting programming language interaction using a user specification approach.

Vivaldi, ViSlang, Diderot, and Shadie are focusing in the generation of data visualizations volumetric having not minded data visualization maps. Superconductor allows the user to create maps because have more expressiveness, but it requires expert skills to implement visual elements. These are different approaches from what we are doing. While Google Maps API, Leaflet and OpenLayers are not so far from ours because they are data visualization maps libraries with high-level abstractions. However, they do not avoid to learn a programming language and pre-processing, insert data and configure the visualization details such as size, icons, layers and data.

Concluding, both DSLs and Libraries are not abstracting from the user all the data visualization pipeline. When using it, users still need to manipulate raw data and transform it in the appropriate format, using external tools or doing it manually. Our proposed DSL provides an abstraction covering the whole visualization pipeline, supporting a high-level language specification for data pre-processing and visualization specifications.

IV. The Proposed DSL

Aiming at facilitating the creation of visualizations for large-scale geospatial data through point phenomena, we proposed an external DSL to provide a high-level specification language. The goal is to be as much as possible closer to the domain vocabulary, supporting a suitable language syntax.

Users that will use this DSL instead of using tools and libraries as Google Maps API, Leaflet and OpenLayers will have some advantages. First, they will not have to know programming aspects like functions, variables, methods and any other web development issue. Second, the user will have a data processing that empowers the data filtering, cleaning and classification automatically as shown in Figure 1. When working with huge files, this DSL allows same operations. Also, we have an optimized file loading in memory to open files bigger than RAM memory available in the system. The third advantage is that this DSL is not linked with a host language. Also, the interface is extremely approximated from the user domain. In this case, for general dotted data visualization maps with a simple interface.

Even we are providing all these features, our DSL also have some limitations. It only allows the creation of already implemented data visualization, and users have to learn a new specification language.

A. DSL Architecture

We internally divided our DSL in three modules as following described. Figure 2 illustrates each module using green arrows to demonstrate input dependencies and blue arrows to express the generation of code or data.

- **Code Analyzer:** This module receives the DSL source code as input to perform lexical analysis, parsing, semantic analysis and code generation. These steps are done by a compiler constructed in C/C++ using Flex and Bison.
- **Data Pre-processor:** C/C++ source files are created after input code analysis. One correspond to the data pre-processor, which is capable to open large-scale files, bigger than available memory, and process these files for applying filters and classification. Then, this pre-processor saves an output file with the data used in the data visualization.
- **Data Visualization Generator:** The DSL Interface module generates a second code that uses the output data from data pre-processor and generates the visualization using HTML, Javascript, and Google Maps API libraries.

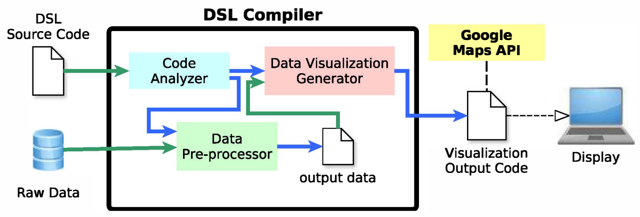

Fig. 2. DSL architecture.

In the following subsections, the DSL Interface, Data Pre-processor, and Data Visualization Generator will be described and explained.

B. DSL Interface

This DSL contributes providing a high-level interface for processing, filtering, classifying data and creating data visualizations maps. It allows to create a data visualization using few lines of code. Being an external DSL, this is limited just to the specifications about the data visualization. Therefore, the user will not mind with other characteristics of host languages like in Superconductor [17]. In contrast, we make this DSL easier to use.

For simplifying the implementation of a new data visualization using the proposed architecture, we considered creating an external DSL. This decision was done because an external language simplify the use since we have the freedom to create an interface similar to natural language.

This DSL language consists of blocks and declarations, as specified in Table I. A block contains declarations that are formed by a property and value. Basically, we have four block names: data, classification, structure and visualization-settings. Declarations may always be used inside a block, with exception for the visualization type that is created in the global scope with data and visualization-setting blocks. In Figure 4 is demonstrated the structure of a declaration by an example of visualization specification pointed by gray dotted lines and in Figure 5 is presented the structure of a block with the main declarations inside using as example a data block.

TABLE I. PROPOSED DSL RESERVED WORDS.

Keyword	Description
Block Names (.a)	
data	This contains all the data declarations.
classification	This is used to declare classification rules.
structure	This names a block with data structure declarations.
visualization-settings	This will determine details for the data visualization.
Properties (.b)	
class, filter	This specify a logic to select/classify data.
date-format	This specifies the date format used in input file.
delimiter, end-register	This is used to inform data delimiters.
file	This receives files location.
latitude, longitude	This specifies the fields containing geo-positioning.
marker-text	This is considered for specifying the text of markers.
page-title	This receives the value for visualization page title.
size	This is used for specifying the visualization size.
visualization	This receives the visualization type name.
Values (.c)	
field	This represents a field when used with properties.
full, medium, small	Values to express sizes.
Operators (.d)	
and, or	Logical operators for join values ($\wedge$ and $\vee$).
contains	Used to verify existence of object inside another ($\in$).
different, equal	Used to apply equality operations ($\neq$ and $=$).
greater, less	Express a logical operation of size ($>$ and $<$).
is, than	Determines a relation between two objects.

In figure 3 is demonstrated the use of *field* value in this interface. The manipulation of data using this interface is done by the *field* value. Considering that a data set stores registers separated in an organized way, using delimiters and end registers characters, we can count each field localization.

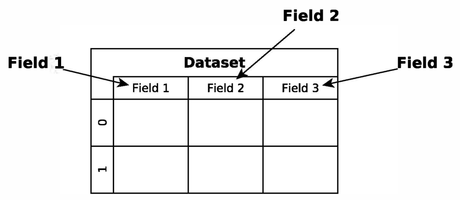

Fig. 3. Field value over data set registers.

This interface have three main elements: a declaration and two blocks. These three elements are specified globally in the source code, as illustrated in Figure 4. The first element consists of a visualization declaration which specifies the type of data visualization to be created. The second element is a *visualization settings block* with declarations of visualization details like required fields, sizes and titles. Concluding the code, an *input data block* with data declarations like filters, classification, data format and delimiters.

Boost this DSL with other types of data visualization in the future is a desired goal. Wherefore, this DSL was built using generalizations seeking to turn easy this objective. An example of language generalization in our DSL is the first declaration in the beginning of code. In this declaration users can inform the desired visualization name as a parameter which enables to change the types of data visualization just changing few lines of code. In Figure 4 is presented an example of a *visualization declaration* element. In sequence, it is constructed a block containing information about visualization settings like required fields[3], page title, visualization size and any information to generate the visual elements.

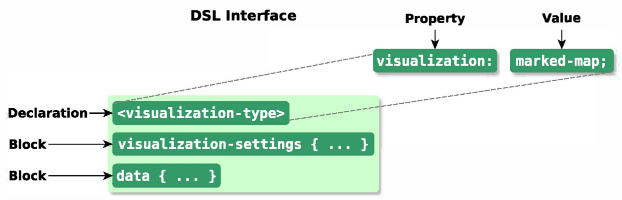

Fig. 4. DSL interface general overview.

Declarations of *page-title* and *size* are inserted in the block represented in Figure 5. The *page-title* property will receive a value. This value must be informed between quotes. The visualization size is informed in the *size* property, which can receive a *full* value for a full page display, *medium* value for a 70% page visualization, and *small* value for a 50% page visualization.

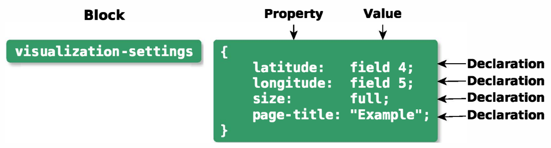

Fig. 5. The `visualization-settings` block example.

The *data block* will contain information about input data and how it will be loaded and processed by the data pre-processor. This block is divided in four elements: *file declarations*, a *structure block*, a *filter declaration*, and a *classification block*, as demonstrated in Figure 6.

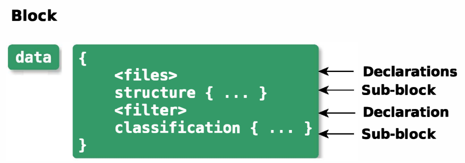

Fig. 6. Input data block tree.

An unlimited number of files can be declared in our DSL. We built in this way because users can use a database separated into many files. This attribute receives a value between quotes that informs the path to a file. Lines 1, 2 and 3 in Listing 1 demonstrate examples of files declaration.

[3]Required fields can change for different types of data visualization. An example of a required field is *latitude* and *longitude* for data visualization maps.

The *structure* sub-block contains information about the file type, delimiter, and end register character. This information is extremely important for the data pre-processor recognize the limits of each register. This block is demonstrated by an example in Listing 2.

```
data {
  file: <value>
  <structure_block>
  <filter_specifications>
  <classification_specifications>
}
```

Listing 1. Database input specification with proposed DSL.

```
structure{
  format: CSV; //Type of input file.
  delimiter: '\t'; //Fields delimiter char.
  end-register: '\n'; //Registers delimiter char.
  date-format: YYYY-MM-DD; //Dataset date format.
}
```

Listing 2. Structure block example.

This interface allows the specification of one or more files to be processed even that it has a limitation. All informed files may have the same format. This limitation exists because data pre-processor uses the same delimiter separators specified in the source code to structure and process all the data files. For example, in a *CSV* file is common use comma as the delimiter and a new line (\n) to end of the register. When working with date, the user also must specify the format used.

An important section in our interface is the specification of logical operations to filter and classify registers. Users can declare a filter to select data and display only useful information in the visualization. A filter declaration uses logical operators to select one or more information of fields. A filter is declared using a *filter* property which receives as parameter a logical operation. Hereafter, a logical operation is declared that when processing over a register add (if true) or not add (if false) this in the output.

We implemented eight logical operators that can be combined in a filter declaration and classification. The operators AND and OR can be used to join logical operations using the other logical operators. We used x for field number, and y and z to illustrate values in Figure 7, where we also demonstrate the relation between the operators that can be combined.

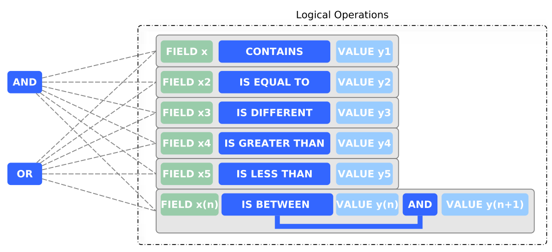

Fig. 7. Logical operators for filtering and classifying.

With these operators, the user can construct filters using logical operations. Figure 8 illustrates the structure of a filter declaration using a simple logical operation example.

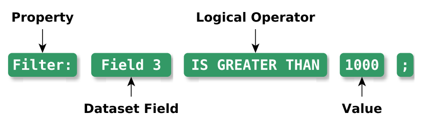

Fig. 8. Simple filter declaration example.

In some cases, the user wants to create classifications. To do so, the sub-block classification were created to allow the user to specify logical operations for the data that will be classified. Listing 3 presents the syntax to create a class. Each class will receive a logical operation, equal to that one used in filters. This logical operation will be applied in all the registers. When returning true, the register will be inserted in the class.

```
classification {
    class: <logical-operation>
    class: <logical-operation>
    class: <logical-operation>
}
```

Listing 3. Database input specification with proposed DSL.

The classification sub-block applied in data visualization maps allows the creation of markers with different colors for each class. We can create classes using the logical operators. Also, the registers inserted in each class will receive a different marker color during visualization generation.

Listing 4 is demonstrating a code example when using our proposed DSL for the Devices visualization type that is generating a map with markers. This data visualization has a device classification. The data inserted in this data visualization map application is from YYFCC100M data set provided by Yahoo Labs[4].

```
visualization: marked-map
visualization-settings {
  latitude: field 11;
  longitude:  field 10;
  marker-text: "<img src='" field 14 "'>";
  page-title: "2014 Photos Classified by Device";
  size: full;
}
data {
  file: "yfcc100m_dataset-0";
  file: "yfcc100m_dataset-1";
  file: "yfcc100m_dataset-2";
  structure {
      delimiter: '/t';
      end-register: '\n';
      date-format: YYYY-MM-DD;
  }
  filter: field 3 IS BETWEEN 2014-01-01 AND
    2014-12-30;
  classification {
    class: field 6 CONTAINS "Canon";
    class: field 6 CONTAINS "Sony";
    }
}
```

Listing 4. Devices visualization example implemented with our DSL.

[4]http://webscope.sandbox.yahoo.com/catalog.php

C. Data Pre-processor

We implemented the Data Pre-processor in five operations: partitioning, structuring, filtering, classifying and output saving. As presented in Figure 9, partitioning is the first function used in this workflow. The partitioning phase allows us to use large data sets in low memory architectures since this separates the entire data set in pieces of data.

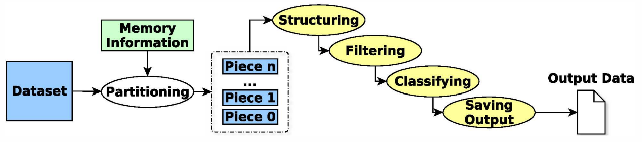

Fig. 9. Data pre-processor workflow.

The pieces of data size shown in Figure 9 are divided considering available memory using the Equation 2. For better understanding, consider the following definition.

Definition 1:
$D = d_0, d_1, ..., d_n$- a group of d_i (data sets)
$F = f_0, f_1, ..., f_n$- a group of f_i (files)
$Fm(f_j)$ = System free memory when processing f_j
$Cm(f_j)$ = System cached memory when processing f_j

Considering:

$$\forall i \exists j (d_i f_j) \tag{1}$$

We calculate the pieces of data with the bellow function:

$$f_j \in d_i | [Fm(f_j) - Cm(f_j)]/2.5 \tag{2}$$

Our *2.5* constant is the size of memory that the pre-processor uses during its operations. For example, whenever 1GB of data loads in memory, during the processing, this information can become 2.5GB due the structuring, filtering and classification processes.

For filtering, we did a study of search algorithms to choice an optimal to perform the registers selection. In this analysis, our study based on Big O notation [18] of each algorithm. Three search algorithms were selected for this analysis: Linear Search, Binary Tree, and Interpolation Search.

For the comparison of the algorithms, we used the time and space complexity considering worst time. As these algorithm may process a huge data set, consider the worst case is appropriate to evaluate them. The comparison of algorithms for data selection was made using the sum of the results of each algorithm. Them, considering that when combining two $\mathcal{O}$ values the greater is the result. We used the greater value to compare each combination with Linear Search algorithm.

Space complexity was considered when comparing these algorithm because the memory space used for them is an important issue to consider. If we are processing a huge data set with a size of 1024 Giga Bytes, an $\mathcal{O}(n)$ space complexity algorithm will load on memory 1048576 Giga Bytes. Obviously, on the actual multicore computer architectures, it can present a memory problem. Then, for this evaluation we seek an algorithm that present a minimum $\mathcal{O}(n)$ for worst time-space, maintaining the same quantity of data.

In our analysis, we found that Linear Search have the greater worst running time of $\mathcal{O}(n)$ since this verify all the elements of the array during the search. On the other hand, this algorithm does not require a sorted array. That choice allows its usage for any input data. Differently, Binary Search and Interpolation Search with $\mathcal{O}(n \log n)$ and $\mathcal{O}(\log \log n)$ need a sorted array to find an element, respectively.

Through this analysis, we can highlight that for ordered arrays the better alternative is an Interpolation Search. For disordered arrays, the Linear Search is more convenient. To certify this, we analyzed this algorithms combining them with some sorting algorithms.

As verified, the time complexity of Binary and Interpolation Search are increased with sorting operations with the minimal complexity time of $\mathcal{O}(k + N)$ using Radix Sort algorithm. The implementation of a binary or interpolation search is harder than the linear search algorithm since it needs the development of a sorting algorithm. We choose a linear search algorithm for executing the filtering step because it will process unsorted data sets in our data pre-processor.

D. Visualization Generator

This DSL have a data visualization generator which produces data visualization code according to the specifications declared in DSL source code. The output file of this module is a HTML file with data visualization done.

Our visualization generator receives specifications from the code analyzer. After, it attaches data-preprocessor output into a HTML/Javascript template to generate the visualization file. Finally, it creates an output file containing a HTML and Javascript skeleton code with the Google Maps API call, the information about title, sizes, and markers, and the data in the Google Maps API required format.

Our DSL does not generate the visual elements of the data visualization. Therefore, we used libraries like Google Maps API in the HTML generated code. We choose this library to simplify code generation because it provides a high-level interface for manipulate maps details and insert data. It also will allow the implementation of other visualizations types without changing the DSL architecture and data pre-processor output. It is possible since the input data format of Google Maps API is the similar used in Google Charts and D3[5].

V. Methodology

We evaluated code productivity and programming effort using the SLOCCount tool. We created four data visualizations with our DSL comparing to three different libraries: Google Maps API, OpenLayers, and Leaflet. We do not included DSLs presented in Section III in this comparison because they do not offer the creation of data visualization maps. Each data visualizations created for this comparison we describe as follows:

- *War* - 2014 photos tagged with ”war” word.
- *Manifestation* - 2014 photos tagged with ”manifestation” word.

[5]http://d3js.org/

- *Devices* - general photos classified by device used.
- *WorldCup* - 2014 photos with ”world cup” words in tags.

For code productivity measurement, we used the SLOC-Count suite, also used by [19] for the evaluation of a DSL interface and by a set of other researches (e.g., [9], [20]). SLOCCount[3] is a software measurement tool, which counts the physical source lines of code (SLOC), ignoring empty lines and comments. It also estimates development time, cost and effort based on the original Basic COCOMO[6] model. The suite supports a wide range of both old and modern programming languages (e.g., C++, Javascript, HTML, and CSS), which are naturally inferred by SLOCCount and thus used for measurement. For our DSL, we selected CSS because it has similar syntax.

In the performance experiment, we used YFCC100M data set [21] as input, which is available from Yahoo Labs. This data set has 54GB of data, divided in 10 files. This is a public multimedia data set with 99.3 million images and 0.7 million videos, all from Flickr and all under Creative Commons licensing. We did a preliminary evaluation of execution time when processing huge quantities of data. For this tests, we replicated the YFFCC100M data set to achieve 100GB of data. We did a measurement of 10GB, 50GB and 100GB to verify the time it takes for processing.

VI. Results

Table II presents the *Source Lines of Code (SLOC)* values generated by SLOCCount. Consequently, we can verify that even with data pre-processing specification in our DSL, the lines of code necessary to develop a map visualization does not increase. In this analysis, we removed the data from traditional libraries that usually is inserted into the source code using Javascript (as previously demonstrated in Section II). The insertion of a new marker inside the visualization can generate until three or four lines of code because it needs to attend the required format, which generates a bigger file. In the proposed DSL, we avoid this issue by using of *Data* block. In this block, we offer features for selecting, filtering and cleaning the data through the use of logical operators, as demonstrated in Section IV. This simplification also reflects in the code productivity measurement, presented in Figure 10.

TABLE II. Code productivity (physical source lines of code).

Application	DSL	Google Maps API	OpenLayers	Leaflet
Devices	22	74	25	79
War	15	20	27	25
Manifestation	15	20	17	25
World Cup	15	20	18	25

Even with the additionally features for data pre-processing, we maintain the code productivity in the four applications when using the proposed DSL. In Figure 10, we can observe that development time are very similar in War, World Cup and Manifestation applications. However, we have an exception in *Devices* application. This occurs because the application has a classification that change icons colors according to the data that will be displayed. OpenLayers require fewer lines of code to implement it, presenting a lower time. The difference between the libraries is due to the javascript implementation to create a new map which configure initial positions and display it.

In this analysis, we measured just the code productivity when creating the visualization applications. The transformation and manual insertion of data in Javascript code, required by traditional libraries, were not considered in our results. In the proposed DSL, we abstract from the user any data transformation due to the high-level interface usage and an automatic data pre-processor. Thus, the code productivity in developing a visualization reduces much more if we consider the user will not handle data manually or look for external tools to automatize this.

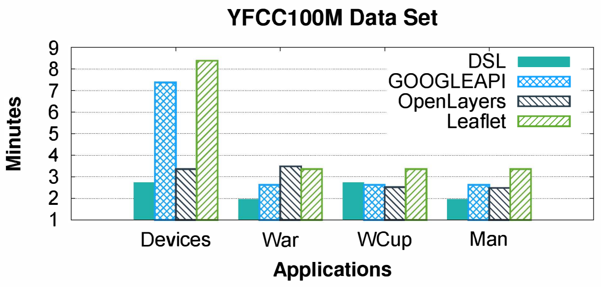

Fig. 10. SLOCCount programming effort results.

We also measured the completion time of data pre-processor. The goal was to verify how long it takes to process a huge quantity of data and generate an output. We could not compare this with traditional libraries once the data pre-processing is done using different approaches, or in some cases, manually. Table III presents this results for data transformation and visual mapping. Data transformation comprehends the total execution time to open, structure, filter, classify the input data, and save the output file. Visual Mapping comprehends the total execution time to generate the visualization and display for the user. We variate the input data in 10GB, 50GB and 100GB of data.

TABLE III. Completion times (seconds).

Size	Data Transformation	Visual Mapping - Google Maps API
10GB	110.4948 (Std. 0.9763)	2.910 (Std. 1.6084)
50GB	544.0506 (Std. 9.4225)	3.2738 (Std. 2.0663)
100GB	1098.9284 (Std. 19.0383)	3.8536 (Std. 2.7584)

If we consider the complexity time of linear search, which is $\mathcal{O}(n)$ like demonstrated in Section IV-C, we can verify a similar behaviour of the completion time in the data transformation as it grows in a linear way according to the input size. Consequently, it is possible to observe an increasing order of approximate 11 times ($11.0494 \times 10GB$, $10.8810 \times 50GB$, and $10.9892 \times 100GB$). We can conclude that as the input data size modifies, the execution time to process the data will keep a relation between completion time and input size, keeping the time complexity in a linear order.

These execution time results demonstrated that as the input data size is modified, the times in visual mapping have a minimal difference. However, data transformation process express a huge difference between the different input sizes.

[6]http://www.dwheeler.com/sloccount/sloccount.html#cocomo

As a consequence, it is important to consider that data transformation process have a higher computational cost when the input size increases. We also observe that the automatic data pre-processing optimizes the visualization implementation. If users develop using traditional libraries, the needing of handle manually a huge quantity of data can take much more time.

VII. Conclusions

This paper introduced the data visualization map problem and provided a new domain-specific language for the geospatial data, supporting big raw data sets. Therefore, we described our description-based language and how we implemented it. During the text, it was possible to point out how much simpler and friendly is our approach with respect to the current scenario. Also, we used a real-world data set to evaluate our DSL, comparing in different visualization types.

Overall, the results demonstrated that our DSL may help data visualization users for gaining insights and extracting information from big data sets. Also, even that we did not measure the effort when users have to prepare the data set, we can highlight that it increases the user's productivity by the possibility of automatically handling raw input data. This abstraction avoids much legwork and complexities on data manipulation such as partitioning, structuring, filtering, and classification.

Moreover, the performance experiments show that we achieved an efficient implementation of the raw data processor as expected by the algorithm analysis (time and space complexities). Also, we identify the opportunity for reducing the completion time. Consequently, we plan as future work to investigate alternatives for taking advantage of the parallelism available on the multi-core architectures, speeding the pre-processing performance.

As mentioned in the paper, we also have in mind to provide more features in our language interface. For example, new visualization types (graphs, treemaps, column and area charts), new logical operators for data selection precision, and offer support for hierarchical data formats such as JSON and XML. Meanwhile, also to include some advanced classification data mining algorithms in our data processor.

Acknowledgment

Authors wish to thank the research support of FAPERGS (Fundação de Amparo à Pesquisa do Estado do Rio Grande do Sul) and CAPES (Coordenação de Aperfeiçoamento Pessoal de Nível Superior). Additionally, authors would like to thank the financial support of FACIN (Faculdade de Informática) and PPGCC (Programa de Pós-Graduação em Ciência da Computação).

References

[1] P. Lyman and H. Varian, "How Much Information?" Berkeley, CA, October 2004. [Online]. Available: http://groups.ischool.berkeley.edu/archive/how-much-info-2003/

[2] J. Manyika, M. Chui, B. Brown, J. Bughin, R. Dobbs, C. Roxburgh, and A. H. Byers, "Big Data: The Next Frontier for Innovation, Competition, and Productivity," McKinsey Global Institute, Tech. Rep., May 2011. [Online]. Available: http://www.mckinsey.com/insights/business_technology/

[3] J. Zhang and M. L. Huang, "5Ws Model for Big Data Analysis and Visualization," in *Proceedings of the 16th International Conference on Computational Science and Engineering*, ser. CSE. Sydney, NSW: IEEE, Dec 2013, pp. 1021–1028.

[4] M. O. Ward, G. Grinstein, and D. Keim, *Interactive Data Visualization: Foundations, Techniques, and Applications*. Massachusetts, USA: CRC Press, 2010.

[5] M. Ghanbari, "Visualization Overview," in *Proceedings of the Thirty-Ninth Southeastern Symposium on System Theory*, ser. SSST 07. Macon, GA: IEEE, Dec 2007, pp. 115–119.

[6] J. Zhang, Y. Chen, and T. Li, "Opportunities of Innovation under Challenges of Big Data," in *Proceedings of the 10th International Conference on Fuzzy Systems and Knowledge Discovery*, ser. FSKD. Shenyang, China: IEEE, July 2013, pp. 669–673.

[7] M. Fowler, *Domain-Specific Languages*. Massachusetts, USA: Pearson Education, 2010.

[8] D. Griebler and L. G. Fernandes, "Towards a Domain-Specific Language for Patterns-Oriented Parallel Programming," in *Programming Languages - 17th Brazilian Symposium - SBLP*, ser. Lecture Notes in Computer Science, vol. 8129. Brasilia, Brazil: Springer Berlin Heidelberg, October 2013, pp. 105–119.

[9] D. Adornes, D. Griebler, C. Ledur, and L. G. Fernandes, "A Unified MapReduce Domain-Specific Language for Distributed and Shared Memory Architectures," in *The 27th International Conference on Software Engineering & Knowledge Engineering*. Pittsburgh, USA: Knowledge Systems Institute Graduate School, July 2015, p. 6.

[10] G. Svennerberg, *Beginning Google Maps API 3*. Berkely, CA, USA: Apress, 2010.

[11] OpenLayers, "OpenLayers Documentation," June 2015. [Online]. Available: http://openlayers.org/en/v3.6.0/doc/tutorials/introduction.html

[12] Leaflet, "Leaflet Documentation," June 2015. [Online]. Available: http://leafletjs.com/reference.html

[13] H. Choi, W. Choi, T. Quan, D. G. Hildebrand, H. Pfister, and W.-K. Jeong, "Vivaldi: A Domain-Specific Language for Volume Processing and Visualization on Distributed Heterogeneous Systems," *IEEE Transactions on Visualization and Computer Graphics*, vol. 20, no. 12, pp. 2407–2416, 2014 Dec.

[14] P. Rautek, S. Bruckner, M. Groller, and M. Hadwiger, "ViSlang: A System for Interpreted Domain-Specific Languages for Scientific Visualization," *IEEE Transactions on Visualization and Computer Graphics*, vol. 20, no. 12, pp. 2388–2396, Dec 2014.

[15] C. Chiw, G. Kindlmann, J. Reppy, L. Samuels, and N. Seltzer, "Diderot: A Parallel DSL for Image Analysis and Visualization," in *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 12, vol. 47. New York, USA: ACM, Jun 2012, pp. 111–120.

[16] M. Hasan, J. Wolfgang, G. Chen, and H. Pfister, "Shadie: A Domain-Specific Language for Volume Visualization," 2010. [Online]. Available: http://miloshasan.net/Shadie/shadie.pdf

[17] L. A. Meyerovich, M. E. Torok, E. Atkinson, and R. Bodk, "Superconductor: A Language for Big Data Visualization." Feb 2013. [Online]. Available: https://engineering.purdue.edu/ milind/lashc13/meyerovich-superconductor.pdf

[18] T. H. Cormen and C. E. Leiserson, *Introduction to Algorithms*. London, England: The MIT Press, 2011.

[19] D. Griebler, D. Adornes, and L. G. Fernandes, "Performance and Usability Evaluation of a Pattern-Oriented Parallel Programming Interface for Multi-Core Architectures," in *The 26th International Conference on Software Engineering & Knowledge Engineering*. Vancouver, Canada: Knowledge Systems Institute Graduate School, July 2014, pp. 25–30.

[20] N. Vazou, E. L. Seidel, R. Jhala, D. Vytiniotis, and S. Peyton-Jones, "Refinement Types for Haskell," in *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*, ser. ICFP '14. New York, NY, USA: ACM, August 2014, pp. 269–282.

[21] B. Thomee, D. A. Shamma, G. Friedland, B. Elizalde, K. Ni, D. Poland, D. Borth, and L. Li, "The New Data and New Challenges in Multimedia Research," *CoRR arXiv eprint*, vol. abs/1503.01817, 2015.