Visualization of Data Flow Graphs for In-Situ Analysis

Master Thesis

by

Jacob Edwards

Submitted to the Faculty IV, Electrical Engineering and Computer Science
Database Systems and Information Management Group in partial fulfillment of
the requirements for the degree of
Master of Science in Computer Science
as part of the ERASMUS MUNDUS programme IT4BI
at the
Technische Universität Berlin
July 31, 2015

Thesis Advisor:
Marcus Leich

Thesis Supervisor:
Prof. Dr. Volker Markl

Author:
Jacob Edwards
Technische Universität Berlin
Berlin, 2015.

Eidesstattliche Erklärung
Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

Statutory Declaration
I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

Berlin, July 31, 2015

Jacob EDWARDS

Abstract

TODO-> write abstract <-TODO

Acknowledgements

Contents

List of Figures

CHAPTER 1

Introduction

1.1 Motivation

IN-SITU data processing is currently extremely popular. In this approach, in order to achieve the minimum possible time in which results are returned, very little preprocessing of any kind is performed. This means that users do not have a very comprehensive understanding of the nuances and problems which may exist in the data beforehand. Any potential pitfalls are likely to only be discovered at a later time, after much time and effort will already have been invested.

Intermediate data sets

Standard statistics such as minimum, maximum, average, or median may help for simple numeric data. However, text data or (semi-) structured data call for different approaches. Aside from knowing what your raw data looks like at the input stage it is also crucial to understand intermediate data sets, i.e. how the different operations affect the data within the data flow.

Directed Acyclic Graphs

It is typical for large scale analysis systems such as Flink [11], Pig [12], or IBMs System S [13] to represent analysis jobs as a series of individual tasks. These tasks are connected into a data flow which generally takes the form of an directed acyclic graph (DAG), which provides a useful visual metaphor for the ordering and dependencies of each task within a job. While this is adequate for describing the process by which data is analyzed, it leaves much to be desired in terms of describing the data itself. In particular, in cases where execution times are particularly long. Thus far, few systems making use of data flow graphs have invested significantly in the area of visual feedback within these graphs. System S provides basic feedback indicating the status of dataset processing without real feedback regarding data features [4], and Lipstick has evolved from a method of

providing provenance models for pig latin queries [12] to providing rudimentary DAG visualization capabilities for Apache Pig in its current development state [12].

Data Set Visualizations

The purpose of this work is to generate visualizations which provide feedback on the intermediate steps within these graphs. These visualizations are proposed in such a way as to be generic enough to suit many types of analysis without modification, and to demonstrate the effects of different operations on the data as well as interesting traits which are inherent to the input data sets in their raw state. Necessarily, to demonstrate this an examination of both common visualizations as well as common analysis tasks are presented, after which the applications of both together are demonstrated and discussed on some representative samples. Additionally, cases where such a solution is either inappropriate or ineffective without modification are presented.

1.2 Structure of this Thesis

Chapter 2 contains a survey of related work
Chapter 3 provides an overview of data formats and visualizations
Chapter 4 examines common Map-Reduce patterns
Chapter 5 demonstrates applications of this work
Chapter 6 describes details of the implementation of this work
Chapter 7 explores potential future improvements
Chapter 8 summarizes the previous chapters and draws conclusions

CHAPTER 2

Related Work

THE FIELD OF DATA VISUALIZATION has existed in some form for as long as data analysis has taken place. The primary purpose of data visualization is of course the effective communication of information through the use of graphics. Across varying fields and time periods, different approaches have been applied to varying degrees of success. Most are familiar with basic forms of information graphics, such as tables or basic charts, but as more data is generated and the economy becomes increasingly information-driven we have seen data visualization expand as a field of study in and of itself.

2.1 Visualization of Data

DATA OFTEN CONTAINS HIDDEN PATTERNS which are very easily understood by humans, but can be difficult to extract using basic statistical or computational methods. A demonstration of this was famously constructed by Francis Anscombe in his 1973 paper "Graphs in Statistical Analysis" [14]. Known as Anscombe's quartet, this example consisted of four data sets containing (x, y) coordinates. Each of these data sets had identical simple statistical summaries (linear regression coefficients, x and y means, x and y variance, and Pearson Correlation Coefficient). When visualized using a simple scatterplot however, each dataset clearly exhibited a unique pattern. Figure 2.1 shows Anscombe's quartet visualized together.

Tufte

General purpose visualization techniques have evolved over the past several decades, but often simple techniques still provide the most effective solution. One of the most seminal works in information display is Edward Tufte's "The Visual Display of Quantitative Information"[2]. This work provided a summary of several different types of

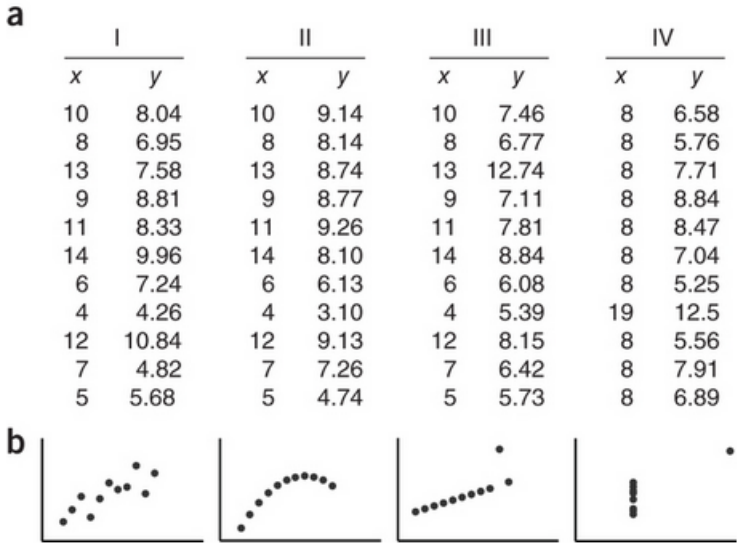

Figure 2.1: Anscombe's Quartet [1]

visualizations applied in many fields, but more importantly it set guidelines as to what makes an effective visualization.

Chart Junk

Many of the key concepts of Tufte's work revolve around the idea of limiting what he called *chart junk*. Chart junk refers to "useless, non-informative, or information-obscuring elements of information displays"[2]. While Tufte acknowledges that using non-data graphics can help to editorialize or provide context for the information being displayed, it is more important to ensure that data is not distorted in order to fit an aesthetic.

Data-rich Visualizations

In addition to limiting non-data information in visualizations, Tufte makes a strong case for the value of data-rich visualizations. Data-rich visualizations are those which include all available information, providing a comprehensive view from which macro trends may emerge. In essence, perhaps at the expense of being able to read individual data points, viewing a complete data set visually may provide insight without need for mathematical analysis. One of many examples of this given in the work is the famous map of central London used by Dr. John Snow to determine the root cause of a cholera outbreak, shown in Figure 2.1. By marking the location of cholera deaths with dots and water pumps with crosses it became immediately clear that deaths were clustered around a central pump on Broad Street. Dismantling this pump quickly stopped the deaths. This provides a clear case where a simple graphical analysis proved far more efficient than mathematical computation would have been in determining a causal link.

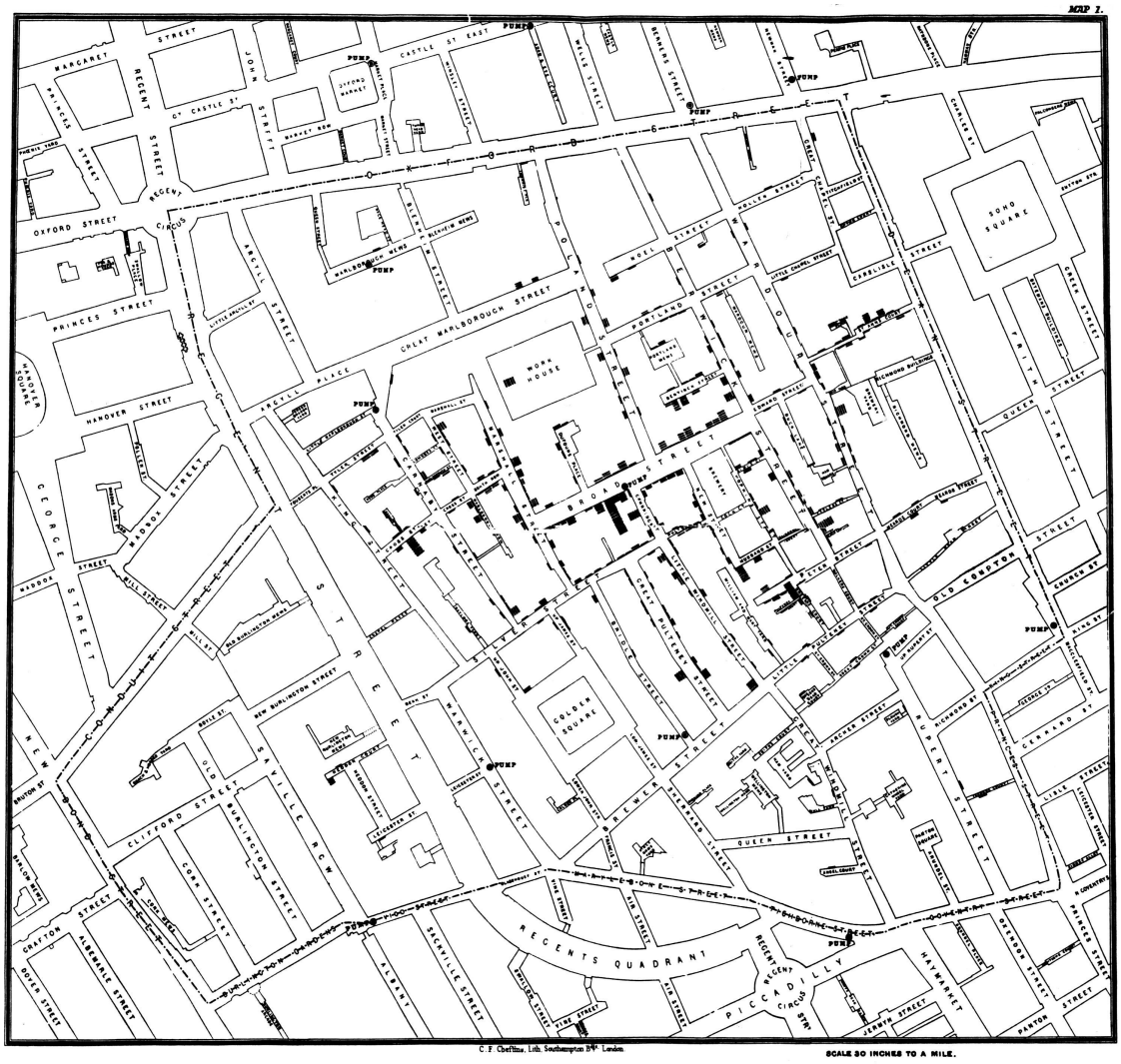

Figure 2.2: The map used by John Snow to determine the source of a cholera outbreak [2]

Dashboards

A more contemporary area of work which is directly connected to digital display is the concept of a *dashboard*. As defined by Stephen Few, a pre-eminent expert in this area, a dashboard is a single-screen visual display of the information required to achieve a specific set of goals. In a business context, this generally refers to key performance Indicators (KPIs). Such a dashboard is typically generated dynamically, allowing for real-time display of data trends as they occur.

Dashboard constraints

In Stephen Few's "Information Dashboard Design" [15] a comprehensive guide to the development of dashboards is given. In particular, specific charts and graphics are matched to appropriate use cases and perhaps more importantly, areas in which some visualizations are inappropriate are defined. Beyond being a discussion simply on visual design, interactivity is discussed. The author notes that although the capability to explore data and perform analysis is available, for monitoring purposes it is more appropriate to not allow such features. Though these analyses are often important, it is more crucial to the purpose of a dashboard to display the data in the form that the dashboard was originally designed for. To do otherwise would risk undermining the purpose, which is a focus on optimal display of key metrics.

Evaluation of Visualizations

Though information visualization has been a very popular research topic for over two decades, there is little in terms of a firm framework by which the success of visualizations can be measured. A review of literature in the area [16] indicates clearly that the literature is mixed on which evaluation approaches produce actionable results, and to what extent these results are accurate. The variables affecting such an analysis include both an examination of the domain in which data is being visualized, and the intended users of said visualizations. Scientific visualizations will not only demand different features than business visualizations, they will often be examined by users with very different levels of expertise. Exigent variables such as user understanding force data visualization to be examined by somewhat subjective standards in almost all cases. It is difficult to determine if there is a number of hours of productivity saved through the use of a dashboard, for example, if the application of the information therein (and therefore its results) is still heavily dependent on unpredictable external factors such as user expertise.

2.2 In-Situ Processing

PROCESSING LARGE QUANTITIES OF DATA has become a common task within many organizations. Data sources such as sensor networks or click streams necessitate handling both massive quantities of information and rapid rates of change. The size of this data presents issues in the efficiency of storage solutions and there are many options

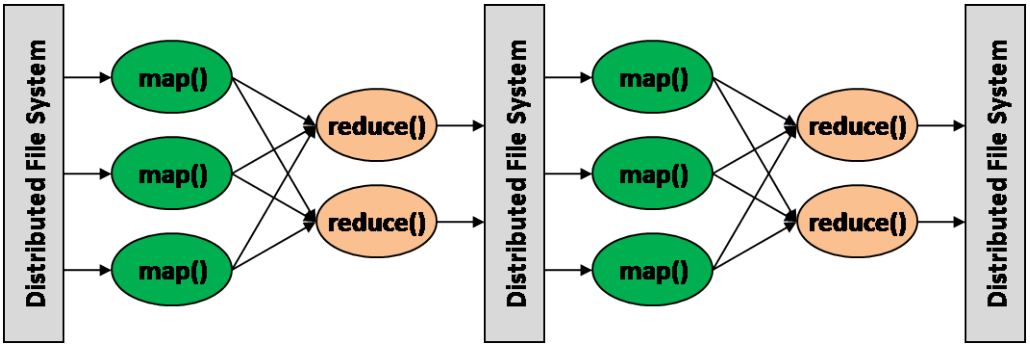

Figure 2.3: A generic data flow graph [3]

for handling such problems [17]. Beyond storage, when analysis occurs on large data stores it is often necessary to apply in-situ processing rather than a more thoroughly controlled approach. In-situ analysis allows for results to be obtained quickly by ignoring much, or all, of the preprocessing that may be involved in an analysis performed on a more controlled data source. Removing preprocessing steps of course increases speed while introducing a number of potential unknown factors. Because in-situ analysis often occurs on data which is unstructured and not stored in a relational format, it fits hand in hand with analysis platforms which operate on large and unstructured data sets.

DFGs

There are many platforms which are purpose built for performing analysis on large data sets, the most common of which are based on the MapReduce model of computation. Conceptually, executing an analysis task in the MapReduce paradigm simply means that the distributed computation task being performed consists of map and reduce operations which are paired to form each step of an overall computation. As more layers of map and reduce steps are added, we are left with a directed acyclic graph of operations which are chained together in a linear way, as can be seen in Figure 2.3. This figure represents two MapReduce steps, each of which is seperated by a write of the data being operated on to the file system. Systems which utilize the data flow graph model optimize these graphs by grouping together and pipelining operations in order to reduce the overhead and cost as much as possible.

Pig

One of the more popular of the data flow graph based systems is Apache Pig [18]. Pig consists of two major components: a language, Pig Latin [19] in which Pig programs can be written, and the execution environment in which they can be run. Pig acts as a high level tool through which users can develop MapReduce applications for execution in a Hadoop environment. Pig Latin provides users with a simple syntax through which sequential operations can be defined, at which point the execution environment compiles

these tasks into Map-Reduce programs for which parallel implementations have already been developed in detail within Hadoop. These sequential tasks are organized into a data flow graph which the system can optimize automatically, greatly simplifying the work of the developer. Additionally, Pig Latin has been designed with extensibility in mind. Developers can write user defined functions in Java, Python, JavaScript, Ruby, or Groovy and then call these functions directly from within a Pig Latin program. Pig was initially developed internally at Yahoo, and quickly became widely applied externally after moving to the Apache foundation a year after its initial development.

Flink

Another platform for large scale data processing is Apache Flink. While Pig Latin provides the interface through which developers can work with Pig, Flink is accessed through either a Java or Scala API. For users who are already fluent in either of these languages, this is very convenient. It allows the same extensibility as seen with Pig, where users can write custom functions for execution within an analysis program, but additionally enables the use of native Java and Scala data types. Using these data types without conversion into the key-value pair data format typical of MapReduce removes one more complication for developers and simplifies analysis programs.

System Differences

Each of these systems have specific traits related to the way their data flow graphs are generated and optimized. Generally speaking however, the graphs themselves are still functionally similar enough that we can attempt to be generic in the way that this work is applied.

2.3 Visualization of Data Flow Graphs

DATA FLOW GRAPH VISUALIZATIONS have existed in some form for as long as data flow graphs have been used in analysis systems. However, their use is almost exclusively applied to examining meta-information such as optimization plans. Relatively little work has been done in generating visualizations which help in the understanding of data, as a supplement to the analyses themselves.

IBM System S

IBM research has developed a stream processing system known as *System S*, which builds processing graphs using predefined operators [13] and has included basic visualization of these graphs [4]. The visualizations show the DAG of analysis operators and indicate whether the operations have completed through colour coding. Additionally, each operator has a small widget which identifies the tuples which have been passed to or from the operator, as seen in Figure 2.4. These tuples can be highlighted in order to show specific data values, and to highlight data dependencies which exist downstream.

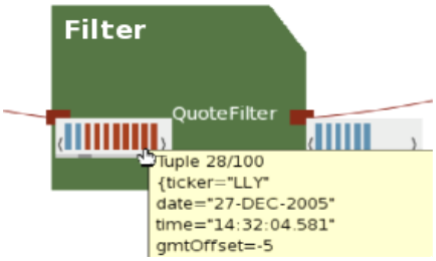

Figure 2.4: An executing operator as visualized in IBM's System S [4]

Retrospective Debugging

This type of visualization exists primarily to support debugging after some failure has been detected post-analysis. It can be seen in Figure 2.4 that there are only ten tuples visible at a single time. Though this number can be expanded, this limitation is here because the envisioned use-case consists of a user scrolling through tuples to identify a single suspected problem tuple. While this is very useful for repairing a problem which is found post-analysis, in cases where this computation is very expensive or the problem is particularly unclear after a failure it may not be efficient.

Lipstick

Lipstick [12], a workflow provenance model framework built for use with Pig takes a similar approach to that of IBM. Lipstick examines the internals of modules within a data flow in order to determine dependencies between parts of a flow. This approach is used for very much the same debugging cases which are expected within System S, with the addition of an added feature allowing developers to query a dependency graph. These queries allow developers to change parameters of the tuples in the graph in order to undertake "what-if" style analyses. Beyond the analysis options introduced through the querying capabilities of Lipstick however, the added visualization features are relatively simple. Like in System S, single operations change colour to indicate status and the tuples being passed to and from operations are identified. In this case the key difference is that the widget for selecting single tuples from System S is replaced with a simple integer iindicating the quantity of tuples moving through a flow. The exploratory capabilities here are left for queries made against the graphs generated in Lipstick.

Flink Plan Visualizer

The approach taken by Flink in visualizing the execution of a job is focused less on the provenance on data and operations and more on the organization of the execution plan as decided by the internal optimizer. Depending on the input sizes and other variable factors in a job, the same program may be executed very differently so that optimal performance can be achieved. Because the development API and the way that programs are executed are independent of one another, it is important that a developer

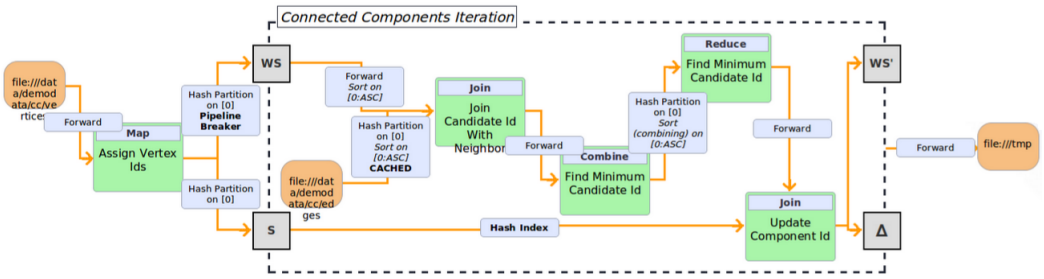

Figure 2.5: An execution plan as seen in the Flink execution plan visualizer [5]

have a mechanism through which they can see the execution order as determined by the optimizer. This is provided through the Flink execution plan visualizer [5], which developers can access through their browsers. A form is provided in which users can submit the execution plan in a JSON format (easily extracted from a running job through the development API), at which point it will be neatly rendered on their screen in a format similar to the example shown in FIgure 2.5.

CHAPTER 3

Visualizations

THIS SECTION AIMS to provide an examination of the methods used to visualize each of the most common types of data in this work. Rather than comprehensively examining all available visualizations, focus will be placed on those data types and structures which are expected to be regularly encountered. These are the data types which are not only most regularly encountered in general, but are particularly applicable to the types of computation scenarios well-suited to analyses in a map-reduce context.

3.1 Numerical Data

NUMERICAL DATA IS UBIQUITOUS when it comes to analysis. Almost all tasks which involve any type of computation will have some sort of summary or statistics to display as a result. This ubiquity has led to a myriad of visualizations being developed for similar tasks, some of which have more merit than others. The key point to consider when visualizing numerical data is to determine the purpose of the visualization.

Category Comparison

Comparing data across several categories is a task which applies to many different forms of analysis. This is best accomplished through the use of a bar chart. Bar charts display discrete groupings of typically qualitative data such as months, product categories, or ages. Rectangular bars are rendered on the horizontal axis, with the bars' heights reflecting the value assigned to their respective categories. The ordering of the bars is often arbitrary, but in cases where the bars are ordered from highest to lowest incidence the resulting chart is known as a Pareto chart. This can help to reveal trends which exist on top of being used for comparison between two individual categories. In cases where the category values are non-discrete, they can be grouped into discrete bins based on semantically sensible ranges. In this case, the resulting chart is referred to as a histogram.

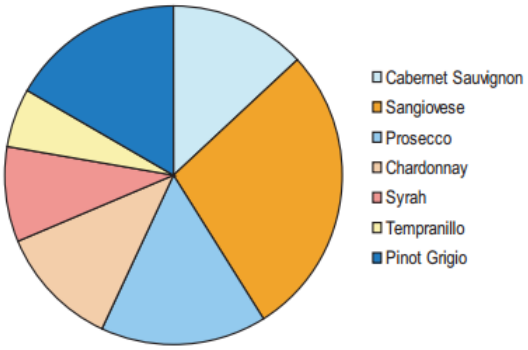

Figure 3.1: A pie chart showing proportions of wine varieties [6]

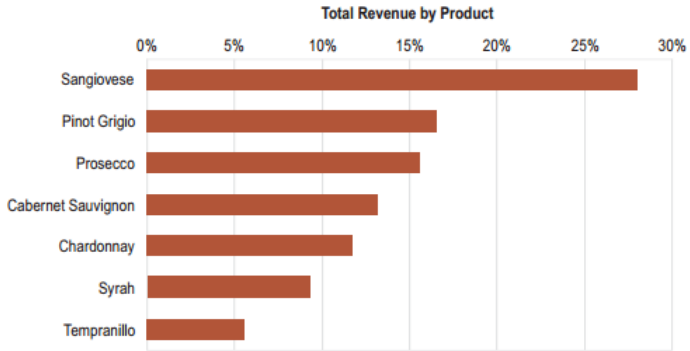

Figure 3.2: A bar chart showing proportions of wine varieties [6]

Pie Chart

Another option for comparing categories is the pie chart. A pie chart is a circle which is divided into wedges representing each category, where the arc length of a wedge reflects the category's assigned value. While these charts are visually appealing, and provide an obvious visual metaphor for parts of a whole, they are generally inferior to a simple bar chart. There are several scenarios in which a pie chart becomes very difficult to read accurately. Primarily, when there are many categories, or when the categories presented are of a very similar size. In such cases, it becomes very difficult to make judgments based on the angles of the various wedges [20]. An example of this is shown in Figure 3.1, in which comparing the blue wedges accurately is quite difficult visually. While it is clear that they are all roughly similar, it is particularly difficult to be certain whether Pinot Grigio and Prosecco have equal quantities, or if one is greater. This could be corrected by adding numerical labels, but the necessity for written numbers implies that a chart may not be better suited to the task than a well formatted simple table.

Pie Replacement

For use in visualizing in-situ data processing in particular, we of course cannot reliably predict the proportion or in some cases number of categories in the data set. As such, it is better to assume the worst case and use a visualization which is more consistantly appropriate. Applying a percentage scale to the y-axis of a bar chart will adequately replace the visual part of whole metaphor provided by the pie chart. Aside from this, their tasks are never strongly divergent, so no other modification is necessary. Figure 3.2 visualizes the same data as seen in Figure 3.1, using a bar chart rather than a pie. In the bar chart, the percentages are very clear and the comparison is much less ambiguous.

Correlation

One of the visual analysis tasks which is best suited to human analysis is the assessment of correlation between variables. In a scenario where a data set exists with two variables, we can place each variable along an axis and mark each record as a coordinate point. Such a chart is known as a scatter plot, and suggests correlation (or lack thereof) based on the pattern of points drawn on the plot. In cases where the drawn points slope from the bottom left to the top right of the chart area, we can infer that there is a positive correlation between the two variables. Likewise, a slope from top left to bottom right implies a negative correlation. A line of best fit can be drawn on top of the scatter plot in cases where the slope is not immediately clear, or simply for clarification.

Non-linear Relationships

While seeing these positive and negative correlations is useful, it is possible to calculate them using simple methods. An even more powerful application of scatter plots is in identifying non-linear relationships between variables. For example, clusters of points are much more easily detected visually in a scatter plot than they would be through the application of statistical methods in the context of an exploratory analysis.

Trends

When examining linear trends in cases where there is a strict ordering of values on one axis, it makes sense to use a line chart. In particular, this is helpful for determining whether there is an increase or decrease in the slope of the line between individual points and through this if there is some causal relationship. Often, in cases where the trend over all data points is more important than any individual measure, sparklines can be applied. Sparklines consist exclusively of the line portion of the chart, and do not normally include axes and labeling. This is generally a design choice, and can be useful in the design of dashboards and other data-rich displays. Because of the ad-hoc nature of the analyses with which we are concerned, normal line charts will be assumed to have subsuming applicability.

Summary Statistics

There are some cases where a visualization more complex than a simple table is unnecessary and perhaps even ill-suited. When there is only a single value resulting from some aggregation, or there is nothing useful to compare resulting values to, for example. In

addition, it may be the case that an analysis is complex enough that a visualization serves to further complicate understanding of the data rather than enhancing understanding. In such scenarios, it often makes sense to simply display the values on their own in comparison with other useful visualizations.

3.2 Text Data

OFTEN TEXT DATA IS PAIRED with some form of numerical summary, and in many cases there is no need for a specific type of visualization for this scenario. This could be true for a data set with products and sales numbers for example, where the product names could easily be switched with an integer key and no analysis value would be lost. However, when there is semantic value which can be extracted from the text we can apply more specific techniques. Particularly, this is true if if we can present the text data itself in such a way that a viewer can assess the basic features of the data more quickly by reading the text than by using a numerical approach.

Word Clouds

The most commonly encountered form of text visualization is a word cloud. Word clouds are a specific form of weighted list which were largely propagated through early blogs and websites as a common feature for exploring tags on posts. There are some examples of these visualizations appearing earlier in printed form [7], but these are generally not for practical analysis purposes. Word clouds can be used to either summarize the frequency with which items occur, or as a categorization method. In a frequency analysis, words within the cloud have their sizes or colours scaled to reflect their associated frequency. Categorization is applied mainly for navigational purposes, with word sizes scaling to the number of subcategories they encompass. Word clouds are often considered sub-optimal for many use cases because they remove context from the analysis and leave too much extraneous information. They still however prove quite practical for identifying flaws or unexpected features of data sets, if not for analysis.

Phrase Nets

Phrase nets [8] represent data to some extent in the same fashion as a word cloud, with the size and colour of a word representing it's frequency in the text overall. The added benefit of a phrase net is that it also shows the relationship between words, providing greater context in later stage analyses. Rather than words floating on their own, they are connected by arrows in a directed graph. The arrows are formed based on a predefined relationship between the two and weighted in the same fashion as the words themselves, based on the frequency with which the relationship occurs. Figure 3.4 shows a phrase net built using the old testament, which connects two words X and Y based on occurences of the phrase "X of Y" in the text. The obvious complication introduced in this approach

Figure 3.3: A word cloud as presented in "Tausend Plateaus: Kapitalismus und Schizophrenie" [7]

is the requirement for a pre-defined relationship. Some simple relationships can be assumed to exist in most text data sets, but similar to a list of stop words it is likely that these relationships will become less useful the more commonly occurring they are. In such a case, we need to obtain a more useful relationship either through user input or some kind of natural language processing techniques. Neither case is particularly well suited for in-situ analyses, requiring significant knowledge of the data at hand or potentially complex meta-analysis respectively.

Word Trees

A somewhat more precise visualization of text data is the word tree[9]. Given a specific pre-selected word, a word tree visualizes the connections that this root word has to other words in different sentences or phrases. As seen in Figure 3.5, a branch is drawn from the root word to each word which immediately follows it in the original data. Likewise, from each of these words the possible children are shown given the prior two words in a phrase. Each word in this tree is drawn at a larger size based on the number of times it follows the previous word. This provides much more context for analyzing an individual word very quickly, particularly in cases such as sentiment analysis where the tone of branching phrases and their weights can quickly identify positive or negative emotion in the data set. A word tree also offers the possibility of interaction, as a user can click any of the child nodes and generate a new tree based on this word. The specificity of this visualization also introduces technical challenges and practical issues for in-situ analysis; in particular the initial input from a user and interactivity (in cases where this is implemented). The initial input from a user implies that some knowledge of the data

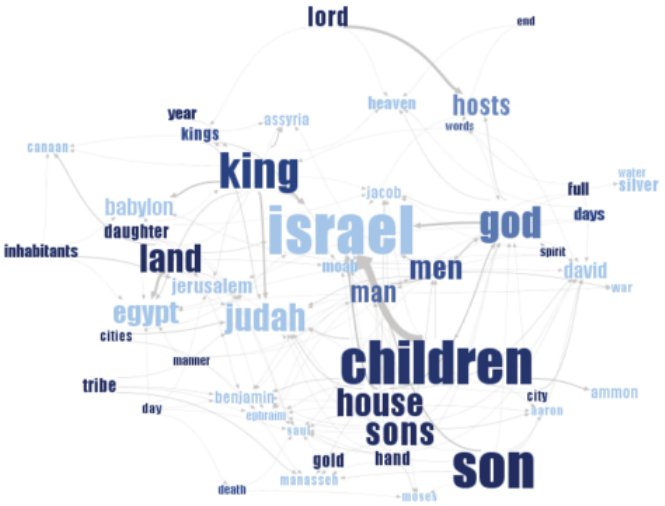

Figure 3.4: A phrase net visualizing "X of Y" in the old testament [8]

set's details are known beforehand to the extent that specific phrases or words are of interest and have been identified, which is likely not the case. More importantly, the method through which these trees are dynamically drawn based on the selection of any node requires re-drawing of the tree and the generation of a new set of sub-trees. This begins to shift the implementation problem from visualization and exploration towards NLP and analysis, and depending on the size of the data set in question introduces a not insignificant overhead in terms of processing.

3.3 Graph Data

GRAPH AND NETWORK datasets are frequently topics of interest for analysts. Particularly, the subset of graph theory known as network theory provides many methods through which analysts can discover useful features of graphs. Network theory assumes that a graph is a representation of asymmetric relationships between discrete objects (as opposed to more abstract definitions as applied in graph theory generally) and has many practical real-world applications. Any area in which real networks between objects occur, such as links in computer networks, social networks, narrative connections in writing, or even molecular networks in biology, can provide a myriad of use cases for analyses that fall under network theory.

Figure 3.5: A word tree visualizing occurrences of "love the" in the King James Bible [9]

Graphs in DAG Systems
KONECT

The KONECT project [21] at the University of Koblenz-Landau has collected a large set of network datasets and provided tools for their analysis. Their collection demonstrates that a very large number of heterogeneous data sets can be modeled as networks, and further that a generic set of analyses can be applied to these data sets if they are represented in a unified way. Though there is a taxonomy of networks based on their respective features (directed/undirected, weighted/unweighted, etc.) the vast majority of analyses are similar if not identical, and differences typically only affect the way in which analysis is performed rather than the analysis result format. Likewise, each of these analyses can be visualized in a straightforward manner.

Distributions

As networks are at their core build of distinct parts, many of the relevant analyses focus on the distribution of features among the nodes or edges therein. Generally, these analyses consist of generating distributions of features across nodes or edges, and thus each can be visualized similarly. Within weighted graphs, the distribution of weights across edges in the graph provides a good representation of any skew or trends in the weighting. An example of this can be seen in Figure 3.6. In cases where the graph is being generated during the execution of some task rather than being provided as input, such distributions can be visualized as a temporal distribution, representing a rate of change in the overall number of edges or nodes at specific points in time, such as can be seen in Figure 3.7. Of course, these distribution analyses can be focused on other

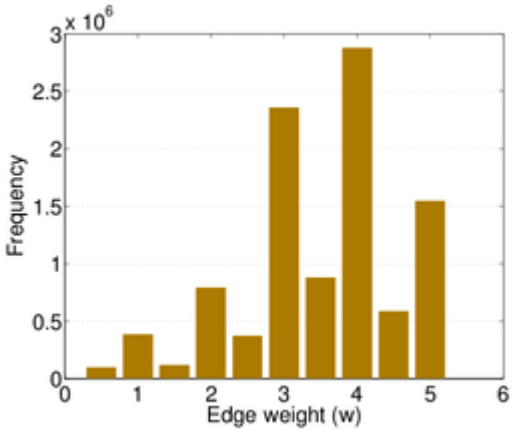

Figure 3.6: A visualization of edge weight distribution [10]

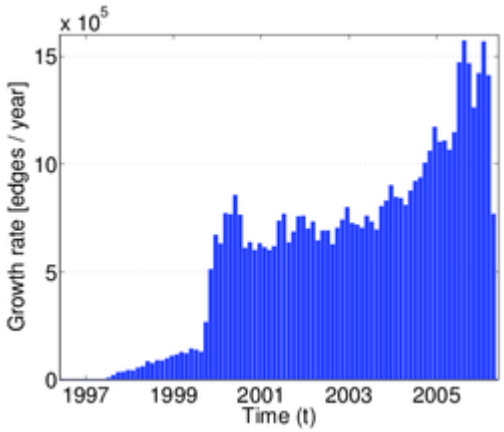

Figure 3.7: A temporal visualization of edge growth [10]

objects within the network, for example degree distributions rather than edge weight distributions. Each type of distribution will reveal some insight into either the nodes or edges of the graph.

Cumulative Degree Distribution

While distributions focus on simple aspects of a network's structure, more complex details can be extracted from the same basic data. For example, a cumulative degree distribution can be extracted to identify the probability that a randomly selected node will have a degree larger than some integer n, as a function of n. Such a distribution can be seen in Figure 3.8. This figure demonstrates that even as we move to more complex forms of analysis, the structure of the output data is still well suited to our basic forms of visualization.

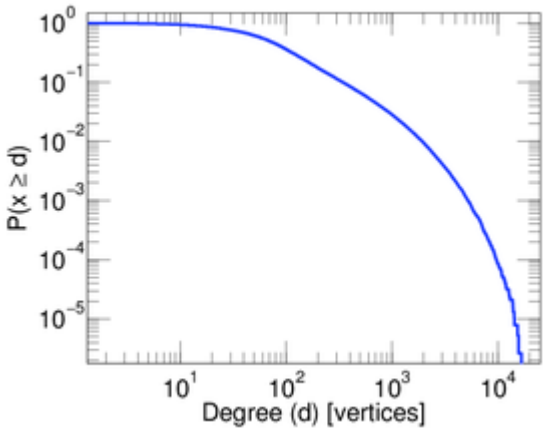

Figure 3.8: A visualization of cumulative distribution [10]

Layout

As graphs are of course a form of structural data, visualizing the structure itself is intuitive. With very small graphs this is easy to do, but as graphs grow many problems present themselves. The most obvious issue is the size itself; a graph featuring ten million nodes will be impossible to visualize in a legible way unless it has very particular features which are accounted for beforehand and visualized accordingly. In the case of even relatively small graphs, it is usually necessary to have some kind of structural information about the graph so that node placement can be handled in a sensible way during visualization. Node placement has to consider not only visibility for users, but also semantic issues such as neighbor proximity and edge overlap. Some generic methods for such problems exist, such as force-directed flow algorithms [22], but it is difficult to predict their efficacy in ad-hoc scenarios where the graph's structure is unknown.

3.4 Summary

LOOKING AT THE SCENARIOS presented in the previous section, it becomes clear that the vast majority of visualization scenarios which would be encountered during exploratory or in-situ processing can be handled using very basic tools. Specifically, these include bar charts, line charts, scatterplots, and simple mutations of each.

CHAPTER 4

MapReduce Patterns

THERE ARE MANY SCENARIOS in which MapReduce can be applied. Because this work is meant to applicable to any MapReduce job, example cases must be selected in order to cover a varied range of analyses and data types. In this case, the analyses chosen attempt to cover the major MapReduce pattern categories as presented in the text "MapReduce Design Patterns" [23]. Of course, in addition to the analysis itself the type of data being visualized is key. As such, these design patterns have been chosen in order to represent the most essential functions of MapReduce analysis, and thus the widest applicable range of input data sets.

4.1 Summarization Patterns

NUMERICAL AGGREGATION TASKS across groupings in a data set are the most common tasks which are encountered in MapReduce analysis programs. This base grouping of data is of course one of the most core functions of the MapReduce paradigm, and thus is often the most straightforward and commonly encountered type of analysis. Because of the simplicity of summarization tasks, they are frequently the first form of analysis performed in the exploration of a new data set. This makes summarization patterns a crucial, albeit straightforward, point of focus for any work concerning the evaluation of unknown factors in data.

Numerical Summary

Numerical summarization is a general pattern for calculation aggregate statistical values across groupings in a data set. Most data sets will be too large for a human to be able to extract meaningful patterns from viewing individual records. Hence, when we are dealing with data that can be grouped by fields in a semantically meaningful way and can be sensibly aggregated or counted we can summarize in hopes of revealing insight.

This pattern is analagous to performing an aggregation after a group by expression in a SQL-like context.

Inverted Index

Inverted indexes are often constructed in scenarios when it is useful to associate some key term with a set of related results within a dataset. This serves to improve search speed by eliminating the need to examine each possible result in a large data set. It does this by pre-restricting the potential results to those which are known to be associated with the search term provided. This differs structurally from the numerical summarization pattern in that the result will be a set of record identifiers mapped onto some search keyword, rather than the relatively simpler group identifier-statistic pairs provided by numerical summarization. Though the actual implementation of the analysis differs, the information which would be useful for basic analysis remains very similar. Namely, the most crucial information here would be the number and nature of results associated with each keyword. This is generally identical to the previous visualization scenario, if we consider the dataset to be visualized as a set of keywords and associated statistical result set metadata (Number of results, average result size, etc.).

Counting

Semantically, counting problems could be considered a subset of numerical summarization. In scenarios where we require only a simple count or summary of a specific field, we could output the key of a record with a count of one and then reduce to generate a final count. The counting pattern instead utilizes the internal counting mechanism of our mapreduce framework to render the use of a reduce or summation stage unnecessary. One can simply create a counter with the ID of the field to be counted and increment by one until logging the result before the end of the execution. An example of a case where this is more efficient than a normal numerical summarization is the classic word count example. As the differences between this pattern differ in implementation rather than goal, the visualization scenarios are likewise identical to those found with numerical summarization.

4.2 Filtering Patterns

FILTERING PATTERNS are primarily concerned with understanding a specific part of an overall data set. As such, all filtering patterns are defined by the fact that they do not alter records in the data set, and that they each return a subset of the original data set. This can be considered analogous to search tasks, in that a set of relevant records is returned based on some provided criteria. All filtering tasks require that data be organized into discrete records.

Sampling

In the context of map-reduce problems in particular, filtering is very useful for sampling. In situations where the data set to be analyzed is too large for processing in full, sampling methods can provide representative subsets. In some cases there are analysis based reasons to perform sampling; such as separating data into training and testing sets for machine learning algorithms. While this alone makes filtering an interesting use case, sampling is of specific interest for this work due to it's frequent application in exploratory analyses. When testing of an analysis job is performed on a large unknown data set, it is intuitive to simply select an arbitrary subset of records to analyze for debugging purposes. This likely provides a skewed view of the data, and an appropriately sampled data set will provide a more representative view of the task at hand.

Filtering

Filtering itself serves as an abstract pattern for the many different types of filtering that can occur in an analysis job. This is of course the most basic filtering function, wherein a subset of the records in a data set are removed based on whether they are of interest or not. In processing systems such as Flink, the purpose is very typically to collect a large sum of data in one place. Simple filtering can serve to either pare some unnecessary data from this sum, or focus on a small set of records or attributes which are particularly important.

Bloom Filtering

Bloom filtering performs much the same task as basic filtering, but with added control in the method through which records are selected for filtering. When applying a bloom filter we extract a feature from each record, and compare that value to a set of values represented by the filter. The primary difference between this and standard filtering are that the decision to filter a given record is determined by the result of a set operation against our filter values. For this approach to be relevant, we must have extractable features which can be compared to the set of filter values, and these filter values must be predefined. It is possible when applying a bloom filter that some records will not be filtered out when they should have been, so they should only be used in scenarios where false positives are acceptable. Such a scenario could occur when prefiltering before performing a more thorough, and much more expensive, robust filtering.

Top N

Performing a top N filter on a data set is of course distinct in that the size of the output data set is known before filtering occurs. Functionally, this is of course very similar to the previous two filtering methods. The application however differs in that there is a clear semantic application of this filter, the collection of outliers. In map-reduce settings this can be a particularly interesting problem as the typical method for accomplishing such a task in another context generally involves sorting the items in a data set, an extremely involved task using MapReduce. This provides additional information about our output,

as we can infer that the output of a top N filter will be significantly smaller than the original data set; otherwise a total ordering is often a more suitable approach.

Distinct Filtering for distinct records is of course self-explanatory in meaning. There are several applications for such a filter, the most common of which is most likely removing duplicate records. In collecting data sets, duplication of records is a frequent data quality issue which can both add unnecessary processing time and skew analysis results.

4.3 Data Organization Patterns

DATA ORGANIZATION problems can present themselves in many ways, and have a wide variety of motivations behind them. With respect to big data problems in particular, the way that data is partitioned, sharded, or sorted can have serious implications for performance. If we consider in-situ processing in particular, there are many cases where data will need to be restructured for further analysis beyond that which is performed in the map-reduce context.

Structured to Hierarchical The structured to hierarchical pattern takes a row based data set and transforms it into a hierarchical format such as JSON or XML. Because MapReduce systems don't care about data format, hierarchical data structures can be immensely helpful in avoiding joins.

Partitioning Partitioning of course separates data into categories. This can be considered semantically similar to a summarization task without any form of aggregation, although the implementation may differ significantly. The major requirement of a partitioning job is to know in advance how many partitions should be created. This can be user provided, or derived from a prior analysis job, in which case the number of partitions may remain unknown to the user. Partitioning becomes very interesting for performance reasons when the partitions are actually sharded across different physical machines in a cluster.

Binning Binning can often be used to solve the same problems as partitioning and is very similar overall. The key difference between the two lies in implementation; binning splits data in the map phase instead of within a partitioner, eliminating the need for a reduce phase. The data structures, and therefore types of visualizations that we would want to see in such a scenario, are identical.

Sorting The total order sorting pattern is of course concerned with the order of records within a data set. While sorting is a fairly standard operation in the realm of sequential programming, it is very expensive and thus less applied in general in MapReduce environments. Achieving total order sorting on data in parallel requires that nor only the data in each

node of the cluster is sorted, but that we can logically join these individual segments of data to form a cohesive sorted whole upon output from the analysis task. The first step in this process is to establish ranges of values within the data set which can be expected to partition evenly across the nodes in the cluster provided. From this point, the data can be partitioned based on the established ranges, and we can sort each individually. The output files from each of these partitions should be appropriately named so that their order within the ranges of values possible is known, then the files can be merged and all output values are in a total order. The cost of this pattern stems from the fact that the data must be loaded and operated on twice; once to determine value ranges for partitioning and then again for the actual sort operation.

Shuffling

Providing essentially the opposite function of the sorting pattern is the Shuffling pattern. This operation serves to randomize the order of rows within a dataset, as well as the order of any attribute values which may appear therein. Applications for such a task are most commonly seen in anonymizing sensitive data sets or performing random sampling tasks. The performance of a shuffling operation is much less expensive than a sort as there is no requirements on the organization of which piece of data is sent to which partition. This also ensures that we will see balanced sizes across all partitions and output files. Aside from these benefits, the shuffling pattern nonetheless still requires that the entire data set be sent over the network and will benefit from the use of many reducers.

4.4 Join Patterns

IT IS RELATIVELY UNCOMMON for all of the data used in a large analysis to stem from the same source. Data can originate from log files, databases, or from a sensor stream feeding directly into HDFS. While joins are simple to perform in other development environments, as is the case with SQL, often requiring only one simple command, in MapReduce environments much of the work must be performed by the developer. Because of the inherent complexity of join operations, there are several useful patterns for implementing them in MapReduce depending on what the specific needs of the scenario are.

Reduce Side Join

The simplest of the core join patterns is the reduce side join. It can be used to execute any of the basic joins seen in a standard SQL implementation (inner, left outer, right outer, full outer, antijoin, and cartesian product) and sets no limits on the sizes of data sets involved. The general use case for such a join pattern is a scenario where flexibility is desired, and a foreign key exists upon which to perform the join. In implementation

terms, a mapper extracts the foreign key from each record and outputs a pair with the foreign key as a key and the entire record as a value. Then, a reducer creates temporary lists for each foreign key value across all data sets, which are then combined based on the desired join logic. This is also the most expensive of the standard join patterns because the foreign key output from the map operation means that no pre-filtering can occur. This cost can be somewhat reduced by applying a bloom filter to the records being output from the mapper. However, with such a filter the reduction in network I/O will be more useful in the case of an inner join than it will with a full outer join or antijoin; which both require all output to be sent to the reducer.

Replicated Join

In cases where only one of the data sets to be joined is large, a replicated join can be applied. In this scenario all data sets excluding the large one are read into memory, thus eliminating the need for a reduce step. The join can be performed entirely in the map phase, with the large dataset acting as input. This is of course a very strict limit set on the size of the small datasets, which is detetrmined by the size of the JVM heap. Additionally, this is really only a valid approach for an inner or left outer join where the large dataset can act as the left data set in the join.

Composite Join

The reduce phase of a join can also be eliminated for larger datasets, through the use of the composite join pattern. This method is limited however by the requirement that the datasets be organized in a specific way. Specifically, all data sets must be able to be read with the foreign key as input to a mapper, they must all have the same number of partitions, and they must be ordered by the foreign key. This is very useful in cases where inner or full outer joins are desired on structured data sources, but in cases such as in-situ processing where guarantees on features of the data set are unknown this is not a practical option for implementation.

Cartesian Product

The last resort in terms of performance is of course a cartesian product. Execution a join by cartesian product is not very well suited to MapReduce, as the operation cannot be parallelized very well and requires more computation time and network traffic than another join. Nonetheless, there are occasions when there is no other option and it must be performed. The most likely candidates for such a join are text document or media analysis where discrete record fields which can be identified as foreign keys are not easily extracted.

Visualizing Joins

While joins undoubtedly perform a vital task in an analysis flow, they do not directly present fertile ground for building visualizations. In the context of performing an in-situ analysis in particular, where we are interested in detecting problems with the data being joined for debugging purposes for example, most problems will be found easily using

other methods. For example, an expected issue with joining data would be some kind of mismatch with the keys being used or an unexpected number of results in the joined set. In the former case, an unexpected value in a key column or a complete mismatch will produce an error in the analysis task code which will not require any special technique to detect. In the latter case, where the join succeeds but some records are unexpectedly lost or included in the join, we will require the application of a pattern from one of the previous categories. Most likely this will be a simple matter of counting records from each set being joined, perhaps by some attribute set, and then visualizing this result similarly to any other encountered in a summarization problem. So, while the task itself is quite different, we can consider the associated visualization requirements to be subsumed by those of the previous categories in all normal situations.

4.5 Meta Patterns

META PATTERNS encompass patterns which deal with the handling of smaller patterns rather than solving particular problems themselves. Because they don't focus on particular problems they don't yield much interesting information for visualization in and of themselves. However, they do provide insight into the way that a large analysis job might be constructed using the previously discussed patterns. This in turn demonstrates the scalability of the previously discussed patterns and by extension the scalability of visualization solutions applied to them individually.

Job Chaining

Perhaps the most intuitive of the meta patterns in MapReduce is job chaining. Large problems are often not easily solved with a single MapReduce job, and thus require a series of jobs to be chained together somehow. In the simplest case, this could mean that several jobs are executed in parallel while others have their input provided by previously completed jobs. This is generally a process which relies heavily on developers, as MapReduce systems are often not equipped to handle more than one job very well and a certain degree of manual coding is required. There are some tools which are being developed to handle this issue, such as Apache's Oozie [24]. Without such tools there are still several options for developers to handle such issues, such as creating a job driver. This is very straightforward, in that a developer simply creates a generic driver task which will call the drivers for sub-tasks in turn when appropriate. Perhaps the most difficult part of such an approach is determining what the most appropriate ordering for execution is and which jobs will require input from some parent job. This approach can also be applied externally by using some kind of script to execute jobs rather than a driver class in the analysis environment itself. The JobControl and ControlledJob classes

form a system for chaining MapReduce jobs in Hadoop, but for simpler applications this may be unnecessarily complicated.

Chain Folding

Chain folding provides a method through which job chains can be optimized further. Because each record can be submitted to multiple mappers and map phases are completely shared-nothing we know that each record will be assessed on its own regardless of grouping or data organization. This means that we can take the map setps of multiple jobs and combine them into a single map phase, significantly reducing I/O load stemming from data movement through the MapReduce pipeline.

Job Merging

Another method which is focused on reducing the I/O costs incurred by jobs is job merging. Job merging applies when more than one job uses the same set(s) of data during their execution. In some cases, if the data set is large enough the initial loading stage may even be the most costly portion of the analysis flow, and is divided for each job that can be merged. There are many complications with merging jobs, not the least of which are the requirements that all keys used in intermediate stages and output formats must match between jobs so that they are both operating on the same data types. A single map function can then be used to perform the tasks of map functions from both of the sub-jobs, adding a tag to output records to identify which mapper task it is associated with. Reducers can then use conditional logic to decide what kind of reduce task to perform based on the tag provided during mapper output. The reduce results can be split to separate destinations at this point for distinct processing on a presumably much smaller set of records.

4.6 Summary

EACH OF THE PREVIOUSLY presented patterns provides a solution to some of the most commonly encountered problems in a data flow graph based analysis scenario. After examining each it quickly becomes clear that while many are directly analysis based, such as counting or filtering, we also see many patterns which are much less reliant (and have less affect) on the visualizable features of the underlying data. These patterns are, by definition, solutions to some of the most commonly encountered tasks in MapReduce analysis. Demonstrating that visualizations are either generally not useful in each case or are attainable using the solution presented in this work serves to demonstrate that this work can be applied to the most commonly encountered analysis tasks and provide the desired outcomes.

Use Cases

Table **??** shows a visual summary of the core visualizations discussed in the previous chapter as matched with the Map Reduce pattern types with which they are likely to be applied. Numerical summary and counting of course both imply category comparisons, which indicate bar charts and line charts being applied depending on expected trends in the data set or a constant dimension such as time. Inverted indexes as a summarization pattern is of course excluded, because this is not a task for which visual analysis of the data would be relevant. Filtering of course can also be applied to categorical data sets, but also provides a more likely application scenario for scatter plots. When filtering data points, visualizing to determine if those records that have been filtered out remove a cluster or reduce noise on a scatter plot is a very common visualization task. Data organization and join patterns are not particularly well suited for analysis using standard visualizations, for different reasons. Visualizing data which has been structured through an organization pattern is heavily dependent on the structure of the outgoing data (XML, JSON, etc.) and will rely on some custom logic and processing in almost all cases. Joins suffer the opposite problem, where the basic exploratory information one would want to extract from a join operation itself is almost always limited to cardinality, and it is probably most efficient to simply output a summary statistic rather than render any kind of visualization. Meta patterns of course relate to program flow, and as such encompass any of the previous pattern types and their related visualizations.

Other Patterns

Other design patterns in the realm of Map-Reduce problems have been introduced, but generally these are related to specific problem domains such as graph processing algorithms or tasks such as message passing and optimization [25]. Though these patterns are useful, they are not particularly applicable to in-situ analysis and thus focus is placed exclusively on the patterns presented in this chapter as being the most applicable. The next chapter serves to illustrate this further with the use of visualizations applied to analysis jobs that fit many of the patterns and categories previously described, and some additional discussion of difficulties and potential for custom visualizations.

Table 4.1: My caption

	Summarization	Filtering	Organizational	Joins	Meta Patterns
Bar Chart	✓	✓			✓
Pie Chart	✓	✓			✓
Line Chart	✓	✓			✓
Scatter Plot		✓			✓
Word Cloud		✓			✓
Word Tree		✓			✓
Phrase Net		✓			✓
Network Diagram		✓			✓
Tree Map		✓	✓		✓
Text Output	✓	✓		✓	✓

CHAPTER 5

Applications

TO CONSTRUCT USEFUL EXAMPLES it is crucial that we consider both scenarios which are both likely to be encountered during a broad range of analysis scenarios, and specific enough to address the basic issues of visualizing unique types of data. The following use-cases discuss the aims of the analysis being suggested and how that relates to the anticipated patterns and visualizations discussed in the previous chapters. The generation of visualizations from an analyst's perspective is discussed, but details of how these are generated are left for Chapter 6.

5.1 Census Data Analysis

THE FIRST CASE WE WILL EXAMINE is an analysis of data extracted from the United States census bureau database. This data set in particular has become a standard example data set used in statistical outlier detection, and particularly in the application and development of machine learning algorithms. It was extracted from the database in 1994, is available online in the University of California Irvine's machine learning repository [26], and was first used in publication in the paper "Scaling Up the Accuracy of Naive-Bayes Classifiers: a Decision-Tree Hybrid"[27] in 1996.

Demographic Data

Demographic data provides a perfect example for most big data type analyses because it can be used in many fields with very little alteration to the methods applied. In social science or political research the study of individuals as would appear in a census is immediately applicable with obvious potential gains stemming from the results of the analysis. Though the census itself may not necessarily be as interesting in the corporate world, capturing traits of individuals in a census is analogous to maintaining a database of employees or customers. Likewise any field which analyses individuals, whether they

be medical patients or users of a mobile application, will apply similar if not identical methods to a data set of approximately the same semantic structure.

Conditional Split

In a machine learning context, this data set is used to predict whether the income of an individual exceeds $50,000 per year. Because this data would normally be partitioned into a training and testing set for use with a predictive model such as a naive Bayes classifier or neural network, it includes a field with the correct response so that testing results can be verified. Thus, we already know whether an individual in the dataset falls into one of the two possible categories (>50K, <=50K) without analysis. We can therefore ignore any prediction and assume a much simpler data flow, a basic conditional split on the category field. If categorical demographic data is to be analyzed in an in-situ context, a reasonable question from an analyst who has not been able to prepare or pre-examine the data set in any rigorous way would be what proportion of records exist across the given categories. In cases where analysts have significant subject matter expertise, a simple visualization of these proportions would be enough to confirm expected results, show an unexpected reality, or imply an error in the quality of data or in the analysis methods.

Split Computation

The actual MapReduce job for such a task is very simple, and consists of an implementation of the numerical summarization pattern. Firstly, the field containing the income categorization flag is extracted from the data source. Then, a flat map function returns a tuple for each record containing the income category and and the integer 1. Following this, we simply reduce by summing the "1" field across each category to determine the totals for each. This is analogous to the standard word count example paired with most MapReduce systems. To perform the visualization, only four lines of code must be added by the author of the analysis task, as seen below:

```
1        Visualizer visualizer = new Visualizer();
2        InSituCollector totalsCollector = new InSituCollector(visualizer);
3        totalsCollector.collect(1, totals, String.class, Integer.class);
4        visualizer.visualizeBarChart(1, "Census Income Categories", "Category", "Count");
```

Visualization Code

The first two lines create a visualizer and in-situ collector, respectively. The visualizer class doesn't require any parameters to be instantiated, and the collector requires a reference to the visualizer class so that it has somewhere to send collected data. The second two lines of code perform all of the actual work in visualizing the data from this flow. The collect method of the collector is called in the third line and accepts three arguments in this case. The first argument is an integer identifier for the collected data set, which can then be referenced later in order to specify which data is to be visualized. The second argument is a data set object from the Flink analysis task in question, and

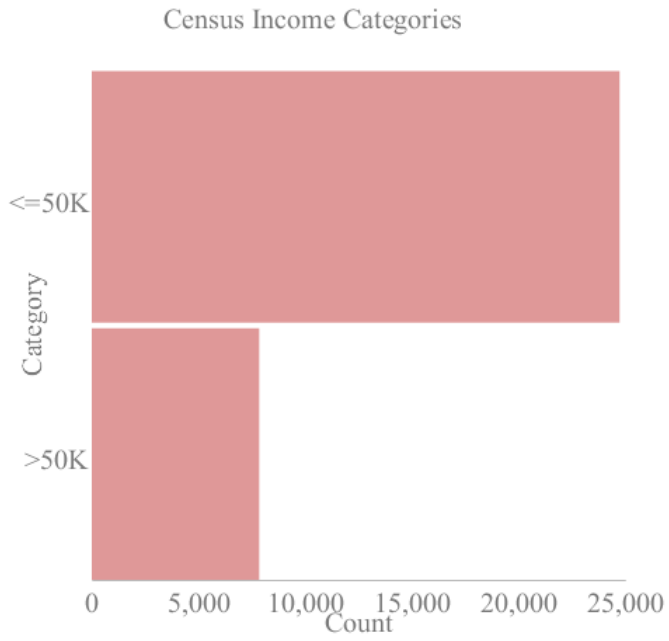

Figure 5.1: A bar chart showing census income category proportions

the remaining arguments are class objects representing the fields contained in the data set "totals", in order. In the last line, the actual visualization function is called from the visualizer. The first, and only required, argument to this function is the identifier of the data set which we collected earlier. Additionally, in this case three string arguments have been provided, which apply a title and axis labels to the resulting visualization. The resulting bar chart is shown in Figure 5.1.

Bar Chart

As discussed in Chapter 3, a bar chart is the most appropriate chart to be applied in the case of category comparison. We can see immediately when looking at Figure 5.1 that those who make less than $50,000 are outnumbered by a rate of roughly 3:1 (in fact, the true proportion is 3.15:1). Given the limited input that is provided in the call to the visualizer, we achieve adequate results in the design of the chart. The labeling is clear and well formatted given the provided input. Even without the provision of labels from the method call, someone with an approximately accurate estimate about the outcome would be able to read the chart without labels. Of course, because we cannot know before the program is run what values to expect, there are some limitations to the way in which we format the results. A good example of this is the x axis. If we desired axis value differences of less or more than 5,000 or a starting point other than 0 it is not a trivial change to make. However, because the purpose of this type of visualization is

focused on getting a sense of data in the in-situ context rather than performing detailed visual analysis, it could be argued that such changes are unnecessary.

filtering

With such a simple analysis task, it isn't unlikely that the developer performing the analysis would also perform some basic drill-down type analyses on the categories in question. In this case, we will filter each of the two income categories and split the records by age. It is of course expected that we will see some kind of trend from year to year in age, rather than strong variance over short sequences. As such, we will apply a line chart as opposed to another bar chart to visualize our expected approximately linear relationship. Figure 5.2 shows the resulting line chart for our filtered set of high income earners. Semantically this shows us an interesting trend in the ages at which people seem to first reach a high income status, and conversely when they age into a lower bracket. More importantly however for an exploratory or in-situ analysis, we see that the data seems to make sense. "Spikes" in the line such as seen between age 40 and 60 are not dramatic enough to strongly indicate a problem in data quality, and in fact the overall bell curve shape of the plot indicates that the sample size for this data set was large enough to reveal a pattern. Of course, a detailed analysis would be required to verify this with statistical significance but for exploratory purposes this is a good start. We can also compare this chart with the records from the opposite filter, as seen in Figure 5.3. This comparison allows us to identify interesting crossover points in the two trends that might be interesting for a detailed analysis. In this case, an interesting point for investigation might be around age 40 where a plateau exists for high earners and the trend for low earners is seemingly unaffected.

5.2 Network Analysis

NETWORK DATA SETS ARE UBIQUITOUS in many fields, as was briefly discussed in Chapter 3. We will examine some of the basic analyses identified by the KONECT project [21], and use the datasets they have provided in order to enable simple comparison of results and reproduction of visualizations which have been proven useful.

Les Misérables

Firstly, we will examine a graph representing data extracted from the novel "Les Misérables" by Victor Hugo. Representing only a single work rather than a corpus of texts, this data set is relatively small. This enables us to examine the features of network data in general, and also those visualizations which relate to layout and will only be applicable with sufficiently few nodes. Within this network, each node represents a character in the narrative of the plot, and the edges represent a meeting between two characters.

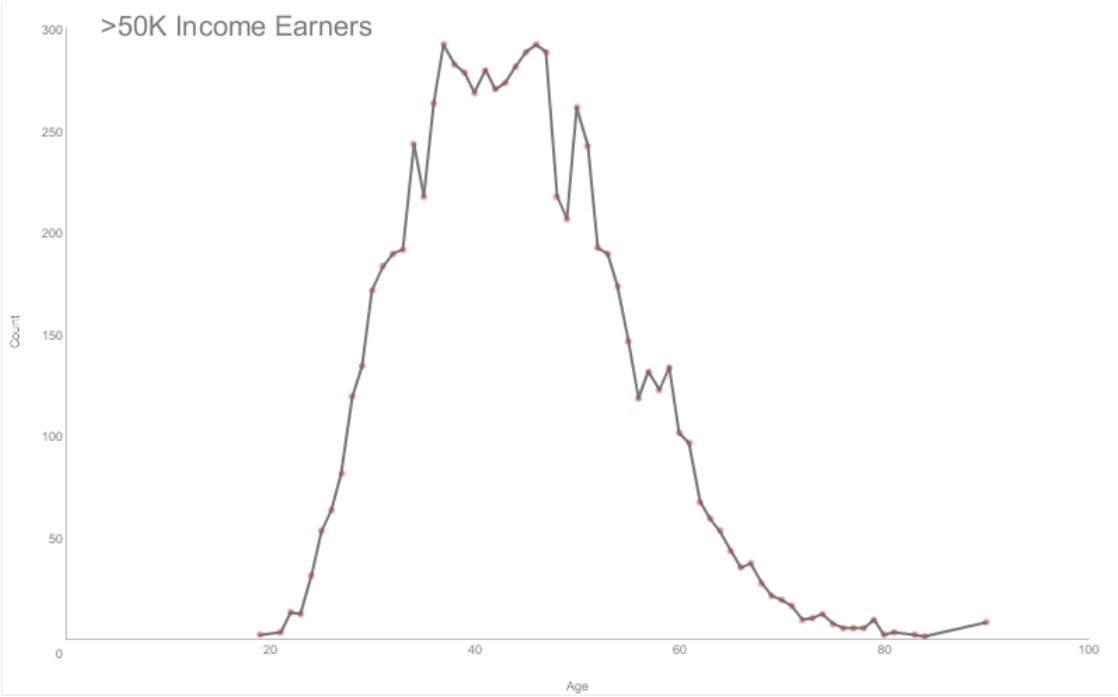

Figure 5.2: A line chart showing the highest earners by age

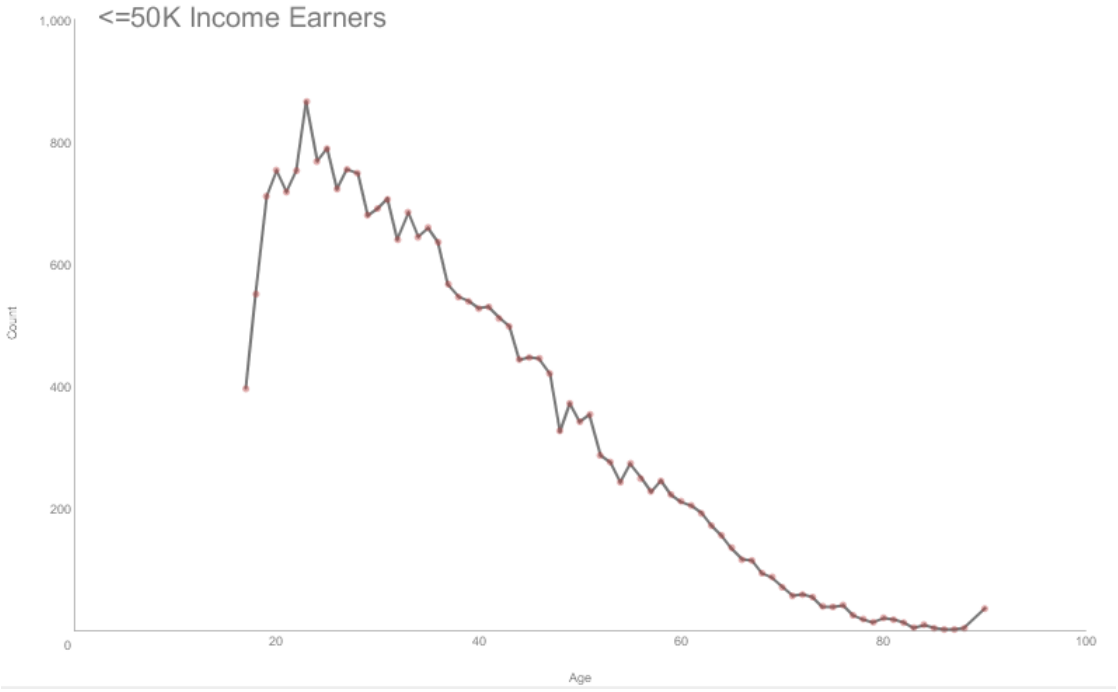

Figure 5.3: A line chart showing the lowest earners by age

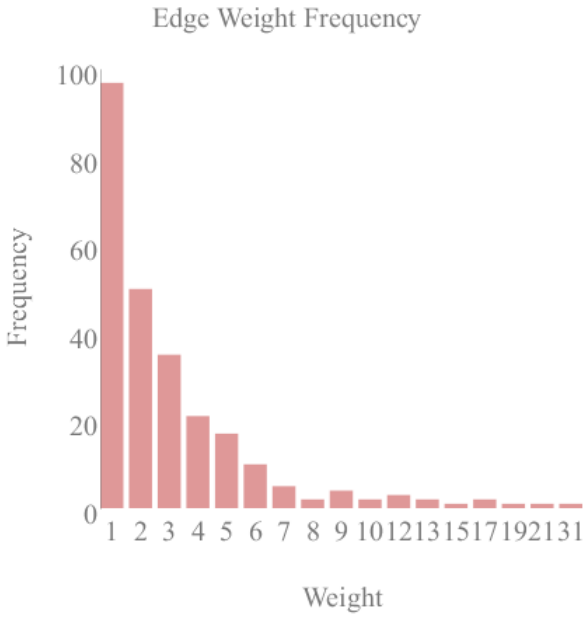

Figure 5.4: A bar chart showing edge weights in the Les Miserables network

Each edge is weighted with an integer, representing the number of distinct times that the characters appear together. In summary, the graph is undirected and weighted.

Edge Weights

Analyzing edge weights is one of the most common tasks that can be performed in network analysis. It is also one of the default analyses provided for all graphs within the KONECT data set, and has been replicated here in the same format to demonstrate that past useful results are replicable. The edge weight frequency graph as generated through this work is seen in Figure 5.4 and that avaiable through KONECT is seen in Figure 5.5. Each shows the distribution of edge weights in this graph, which in this case forms an approximately smooth decreasing curve. This is sensible for the graph in question, indicating that there are more character within Les Misérables who appear very infrequently, and thus encounter few other characters, than there are major characters for whom the opposite is true. In many situations, this kind of trend in edge weights may seem like an obvious conclusion, but there exist many scenarios where this is not the case beyond simply detecting errors in the data itself.

Wikipedia Conflicts

An example of a graph which displays a very different trend is the Wikipedia conflict graph [28]. This dataset was originally extracted for the purposes of examining structural similarities in social network graph data [29], and represents wikipedia users as nodes. Each edge in the graph represents an interaction between two users in conflict, an

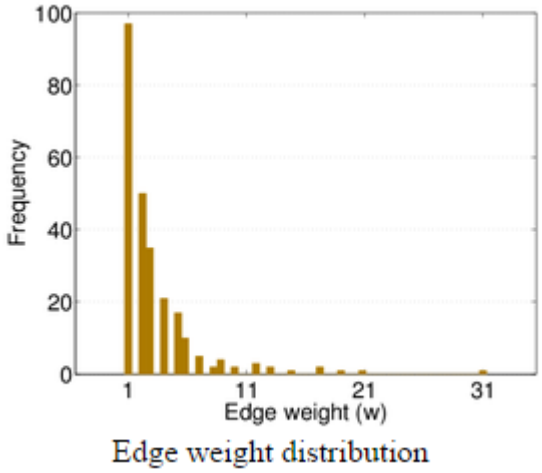

Figure 5.5: A bar chart showing edge weights from KONECT

edit war being the typical example, with graph weights representing the nature of the interaction. Negative weights are associated with negative interactions, and the opposite for positive weightings. Examining Figure 5.6 shows that we see an approximate bell-curve, with a gap in the center indicating that no neutral interactions were coded. It should be noted that due to the increase in the number of categories, the automatic formatting of the x-axis has caused the labels to run together somewhat, but the overall trend and important values are still easily identifiable.

Degree Distribution

Though edge weights are obviously specific to weighted graphs, there are other straightforward methods for assessing the structural elements of network data that apply to all graphs. A very common method of this is the calculation of degree distribution. Once again, a side by side comparison of the result generated by this solution and that given by KONECT is shown in Figure 5.7 and Figure 5.7 respectively. Here, the difference in the results of each visualization is stronger because of axis scaling. The KONECT visualization uses logarithmic scaling for both axes, but due to the general purpose design of the scatterplot visualization used in this work normal scaling is applied. Due to the limited size of this graph in particular normal scaling actually provides a clearer picture of where exactly points sit, but as the data set size is scaled up the number of points minimizes the effect that scaling has on legibility.

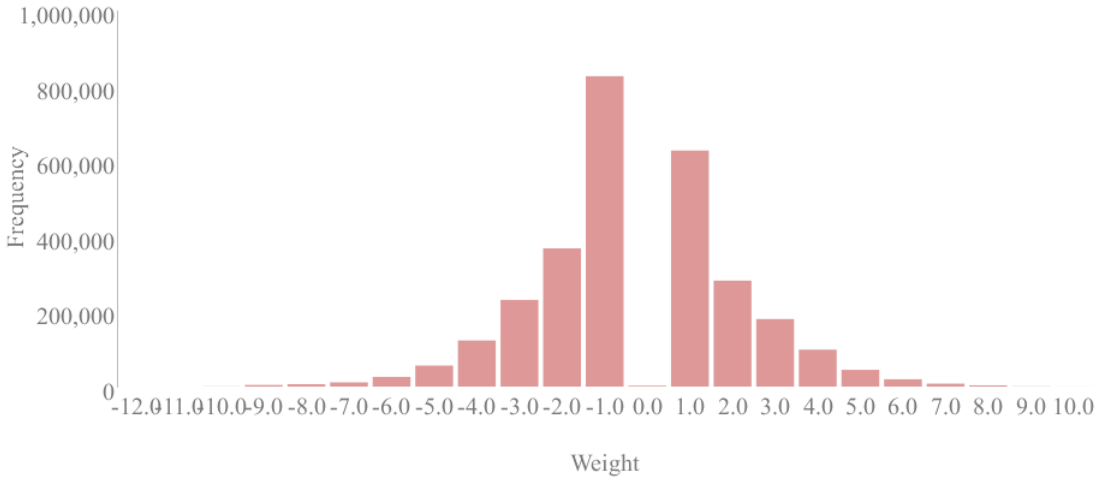

Figure 5.6: A bar chart showing edge weight distribution in the wikipedia conflict graph

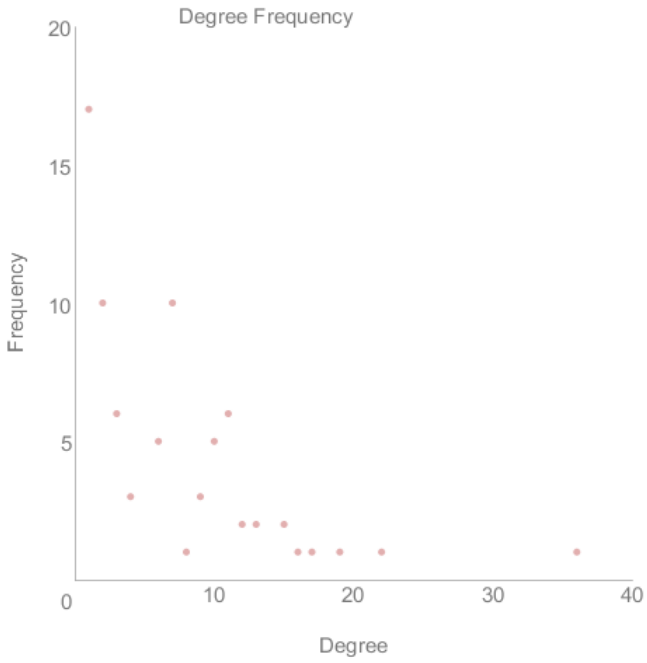

Figure 5.7: A scatter plot showing degree frequency in the Les Miserables network

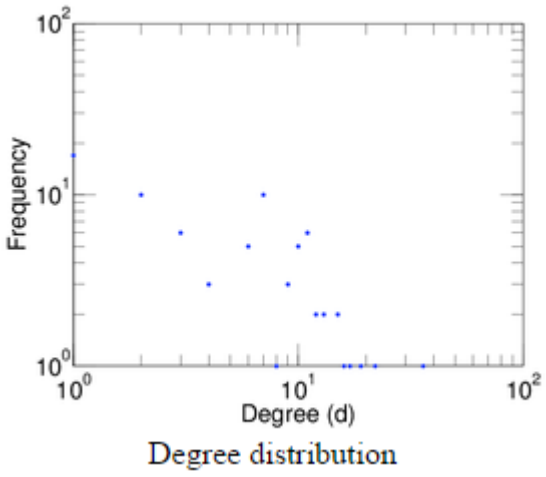

Figure 5.8: A scatter plot showing degree frequency from KONECT

5.3 Classification

AS ONE OF THE CORE TASKS IN EXPLORATORY DATA MINING classification provides a scenario which would very likely rely on in-situ data processing. The data set to be examined is the "Iris Plants Database" [30] which first appeared in 1936 as an example of discriminant analysis. It consists of four measurements: the width and length of both the petal and sepal of three different species of iris. Based on its initial use in developing a model which distinguished different iris species, it has since been used as a standard test case for various classification techniques. If this data is examined from the perspective of an analyst with limited prior knowledge of the features of the data (apart from the species included) visualizations can be used to limit potential classification methods.

Scatter Plot

A basic visualization of this data such as in Figure 5.9 immediately indicates that there are two distinct clusters in the data set. Additionally, it can be seen that some of the points in the scatter plot have darker colouring than others. This indicates identical records in the source data set, with darker points indicating more duplicates. Because axis scaling is based on the values present in the data set rather than some pre-determined range, they are well distributed across the layout of the plot at the expense of what is arguably an unorthodox axis origin. The two cluster shown isolate on of the three species in the data set, but do not separate the remaining two visually. This is somewhat interesting, but is limited in that only two variables are seen at once, perhaps hiding some relationship that exists between another combination of our iris features. As we expect three classes of

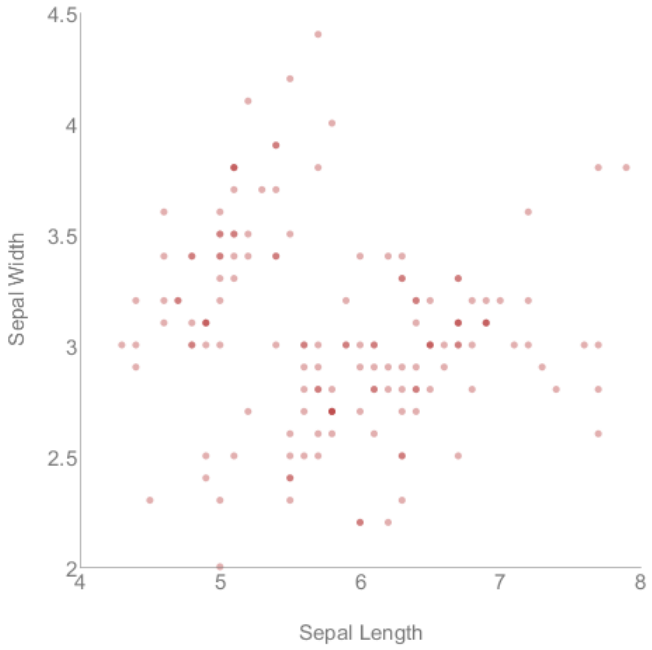

Figure 5.9: A scatterplot showing iris sepal data

data within the set, it follows that we should hope to see three approximately separate clusters in one such comparison.

Matrix Plot

A standard method for examining the relationship between multiple variables is to simply visualize each combination in a grid, in this case as a scatter plot matrix. Such a visualization can be seen in Figure 5.10. While the fine grain details in such a visualization are somewhat more obscured in comparison to the individual scatter plot, we can now easily obtain an approximate idea of the relationships between each variable in our data set. The matrix is composed of a square grid with dimensions equal to the number of variables to be compared, with the main diagonal of the grid representing the variables themselves. The remaining squares in the matrix are filled with scatter plots; where the x-axis of each represents the variable with which it shares a column, and the y-axis the variable with which it shares a row. The relative size of the plots will of course be scaled to the number of variables present, but may be impractical with larger numbers of variables. The labeling and titles are less important here as we are focused on broad visual patterns more than specific statistic, similar to the case with spark lines. Also, because labels for the variables do not exist in the Flink data set being analyzed in this case, placeholder names are given to each variable. It is assumed that either the user is aware of the order in which they were provided to the collector and can infer

their semantic meaning, or that the specific variable names themselves are unimportant relative to the patterns exhibited.

Classification Method

In this case, we can see that across all plots in the matrix only two clearly demarcated clusters exist. This means that regardless of which variables are chosen for comparison, we are unlikely to be able to separate all three species of iris using a simple linear method. An analyst faced with such a figure would be likely to apply a supervised learning method if at all possible, or perhaps a more complex unsupervised method such as a linear principal component analysis.

5.4 Custom Applications

IT WAS NOTED IN THE PREVIOUS CHAPTERS that several of the presented patterns and data types presented specific problems which would require a solution that couldn't be handled through generic methods. One such case that presents very clear difficulties for visualization in an in-situ environment is hierarchical data, such as that which would result from the application of many of the data organization patterns from the previous chapter. Though hierarchical data is generally presented in a familiar format such as XML or JSON, visualizing this is not an insignificant task.

Treemaps

A relatively new visualization which is purpose-built for displaying hierarchies is the tree map. A tree map is a very space efficient method for displaying a hierarchy using a set of nested rectangles. Nesting of course occurs based on levels in the hierarchy being visualized, and layout and coloring can be performed based on features of the data. Though area based visualizations such as mosaic plots have existed for a very long time, the generation of a tree map is dependent on a recursive algorithm for generating tiles. Though the first instance of such an algorithm dates back over 20 years [31], attention to the ways in which tree maps could be applied to varying types of data and analyses became much more popular several years later. In displaying hierarchies, the way in which hierarchies were visualized colored and labeled varied strongly depending on how many dimensions were contained in the data and what the purpose of the analysis was [32][33].

Specific Implementations

Even though there are many implementations of tree map visualization algorithms, and some layouts may be better suited to some tasks than others, it can still be useful to visualize any hierarchy to demonstrate basic structural features. For demonstration purposes, Figure 5.11 shows a tree map generated using a processing library developed for use in visualizing spatial hierarchies [34] [35]. Using the tools provided in this library

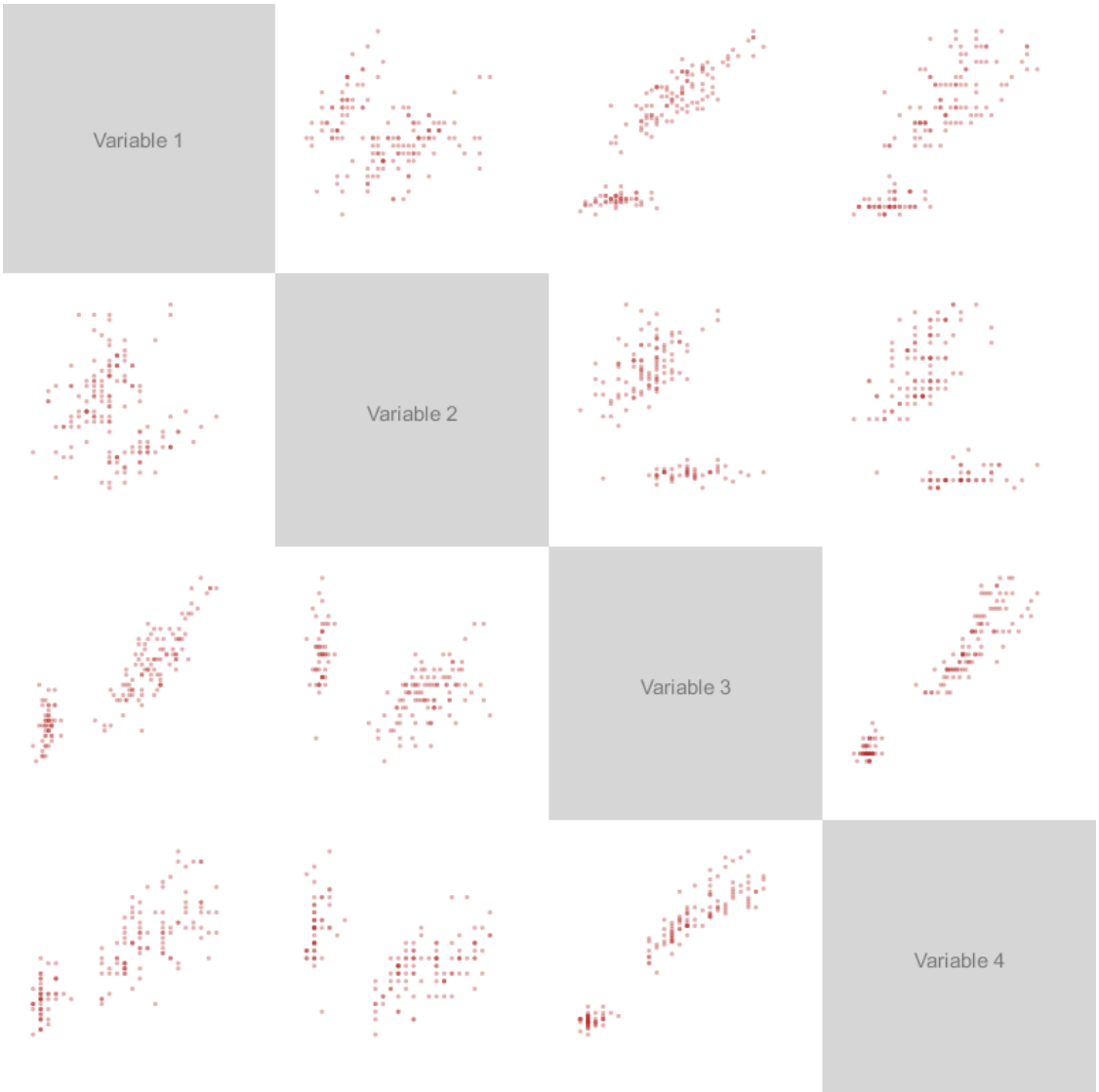

Figure 5.10: A scatterplot matrix showing iris data across all variables

for handling layout, coloring, and labeling I have constructed a simple visualization class which parses a data set into a format which is digestible and draws it. The data being visualized represents wikipedia contributors by country and continent.

Input Format

This input data handling is where the problems begin to emerge. Key to the interpretation of data for tree map generation is knowledge of the number of layers in our hierarchy, and to which layer each element belongs. This means that it is impossible to simply parse an XML file, for example, row by row into a more mappable format as we require the context of the rest of the structure. To obtain this context we may have to examine the entire structure, which in many cases may be large enough that this introduces a significant processing overhead to the visualization. The necessary input format to most tree mapping methods is exacting enough, that it is unlikely a method generic enough for general use could be developed without unacceptable cost. The input requirements of the method used in Figure 5.11 for example require a comma delimited file in which each row represents a leaf node, and specifies the full hierarchy path to that leaf as well as its depth. It can also accept an esoteric XML format called TreeML, which was proposed for use in a data visualization research competition [36] and has since fallen into disuse.

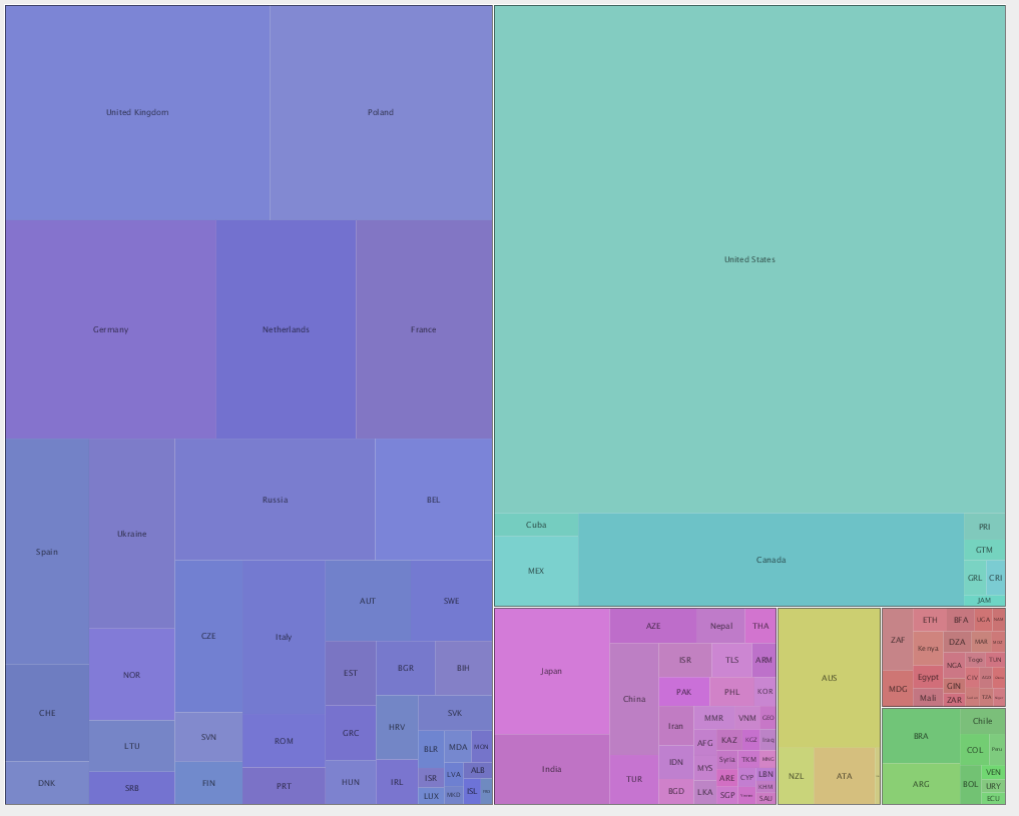

Figure 5.11: A tree map

CHAPTER 6

Implementation

THE PROPOSED METHOD for implementing an in-situ visualization system is comprised of several vital parts. Although the output visualization is key from a user perspective, there are important factors to be considered in the way that data is collected and how this method fits into the overarching analysis system.

6.1 Overview

THE CORE DEVELOPMENT PORTION OF THIS WORK is based on the classes which generate visualizations using a Flink execution plan. Firstly, there is an In-Situ Collector class which has the sole purpose of collecting data sets and/or summaries of data sets as they are run through the Flink analysis task. After data has been collected, the Visualizer can perform various visualization tasks based on the datasets which it has been provided. Figure 6.1 shows the basic structural parts of this development.

Visualization Classes

While the aforementioned two classes perform the bulk of the mechanical work, the visualizations themselves each require their own specialized classes which can be invoked generically from the Visualizer. For standard visualizations such as a histogram these classes largely handle the translation of data sets into a more easily digestible format which can be passed to pre-existing classes or visualization libraries. In more complex and specific scenarios such as generating a scatter plot matrix, 'sketches' have been written in the Processing visualization language. These sketches can, with some minor modifications, be used within java projects and then drawn using the java swing toolkit.

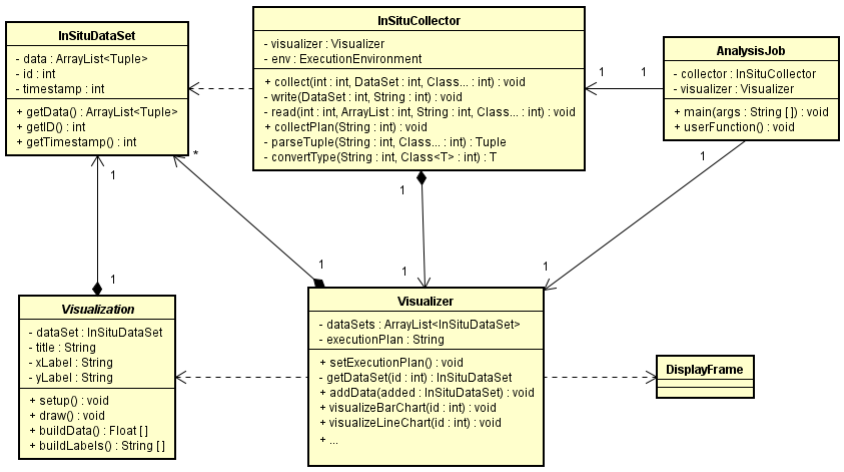

Figure 6.1: A UML diagram of the core classes

6.2 Data Collection

DATA TYPES IN FLINK are analyzed by the optimizer to determine the most efficient execution strategies. In order to make this process simpler, Flink places limits on the types of data which can be used. There are four categories of types: General objects and POJOs, Tuples, Values, and Hadoop writeables. The handling of each of these types must of course be considered when data is being collected from an analysis graph.

Tuples

Tuples are used to represent composite data sets, and are composed of a set length list of fields of various types. Tuples can include any valid Flink type as an element, including further tuples. One of the major benefits of using tuple types is the ability to use built-in functions to navigate through the tuple values. Specifically, these functions allow the selection of specific fields as the key for operations and more generally allow the navigation of tuple fields using string expressions.

Data Collector

The data collector class acts as a simple addition to a pre-existing analysis program in Flink which collects data as it passes through operations. A single collector object exists for a given analysis flow, and collects data at a specific point with a single added line of code calling the collect method. As the data is collected, a user specifies an integer identifier which marks the data set and can be passed to the visualizer later in the task so that the appropriate data set is drawn into whichever visualization is selected.

Collect Method

Each time the collect method is called, it sends a new dataset to the central visualizer class. This method accepts a dataset, identifier, and data types as its arguments and

writes this dataset to memory in a format which can be read by the data collector. Each segment of data within a data set object (corresponding to each distributed partition in the overall data flow) is written individually in the same directory of the file system. The writing of this data constitutes a data sink within the execution plan, and thus after each write phase of the data collection process the execute command is called from the analysis program's execution environment. This execution forces the data to be fully recorded before attempts to visualize or collect new data can occur. The data collector then reads this data into a new dataset outside of the original analysis flow's execution environment.

Type Erasure

Reading the data back into the visualizer after the write operation has occurred is a source of more complication than the collection itself. During the execution of analysis jobs, the compiler will erase types and operate exclusively with generics. This means that when data is extracted from the execution environment for writing and subsequent reading in the collector, the specifics of the original types are not available. For simple data sets, for example exclusively numeric data, it is possible to simply use pattern matching to determine the types of the written data when it is re-read into the collector. However, when handling data sets in which the type is not clear simply from examining a single record this method fails. An simple example of this would be a text data set in which a record consists of a single word; if any numbers such as years or statistics are listed in the text it becomes impossible to determine type from one record alone. Though an issue as simple as this could be avoided through some kind of look ahead sampling method, there is of course also the issue of general data quality. This is particularly unpredictable in the case of in-situ processing. As such, in the collect method the user is required to provide a list of classes which correspond to the respective tuple attributes in the collected set. This allows the collector to assume the type of read data, and eliminates any possible ambiguities. After the read operation occurs, all written data is erased from the file system.

Data Sets

After data is re-read from the file system by the collector, it is added to the visualizer object. A custom data set class exists for the use of the collector and visualizer. This class is very similar in core function to the data set class which is native to Flink, but allows for the tracking of additional metadata which may be useful for debugging. This information could include timestamps, tags referring to specific operations in the analysis flow, or other semantically relevant information. These datasets are always initialized to contain a list of tuple type objects. As a tuple can of course include any item of a basic type, this implementation will create a tuple of any general object in order to simplify data set operations. For example, if a single integer field is passed through

the initial analysis flow, the data set generated in the visualizer will consider this as a tuple of size one which contains an integer.

6.3 Visualization

DEVELOPING VISUALIZATIONS in software is a matter of both design and engineering. Finding an effective way to build visuals is often as important as the visualizations themselves. In building the visualizations in this work, different languages and libraries have been applied in order to demonstrate the most important applications of this work in a concise manner.

Factory Pattern

The factory pattern is applied in the visualization process, in that a creation method is called from the visualizer at the analysis program level, but a concrete creation method is called from within the visualizer which allows different implementations of the same type of plot to be drawn. This is important for expansion and adaptability of the software, as shifting use-cases and user requirements may require alterations to existing visuals or a complete redesign of some features. This could relate to something as simple as aesthetic changes, such as modifying the way that charts are coloured, to completely redefining the structural elements of charts and their handling of different data types. Additionally, this allows for a combination of custom written classes and libraries to be used if needed so that difficult visualization tasks can be completed.

Processing

Processing is a language which was initially developed as a teaching tool for computer programming fundamentals which utilized visual arts as a context. It was first released in 2001 as a project of the MIT aesthetics and computation group and has since evolved into a professional level tool for visual programming. The primary advantages of using Processing as a tool for portions of this work are it's ease of use, and compatibility with the rest of the development environment. As it was initially intended as a learning tool, the structure of a processing program is often very simple when compared with something similar generated using only java for example. A single program in Processing is referred to as a "sketch", referring to both the artistic nature of the language and the typical simplicity of it's application. In addition, processing code is compiled into java which simplifies the integration of the two and ensures portability.

Libraries

The City University of London's Graphical Information Center provides several useful libraries for performing visualization work. In particular, to aid in the development of work which utilizes processing sketches. The visualizations in this work have been built using utility classes from these libraries in many cases. The libraries provide aesthetic

functions for creating charts, such as spacing of axis labels, colouring functions, and bar chart shapes. These functions provide the skeleton for all basic visualizations, upon which logic for handling data and adjusting the specifics of formatting have been added.

Swing

Outside of the visualizations themselves, the work of creating frames and navigation is largely handled through directly using java's swing visualization toolkit. A basic JFrame is generated whenever a visualization is created, upon which the visualization is drawn. This frame simply scales to the same size as the visualization it is meant to contain, and acts as an intermediary between the visualizations themselves and the visualizer.

Visualization Class

Each visualization class can vary greatly, as their only requirements are that they can be written to a swing panel and that they accept a data set object from the visualizer. All visualizations which have been implemented so far simply consist of setup and draw methods, as well as any utility functions which may be required. The utility functions generally consist of taking the data from the data set and formatting it for consumption into pieces of the visualization such as the axes or bars/lines/etc. In the most complex case,the scatter plot matrix, the visualization class actually divides the data set into segments and generates sub visualizations for each square in the matrix. It is possible for any visualization to rely on a number of utility classes and libraries as long as the top level visualization class still meets the basic requirement of fitting into a single frame.

Adjusting Visualizations

The currently implemented visualizations have a certain degree of adaptability built in. For example, axes and overall size of elements will scale based on the data set to be visualized. However, there are some features which are not automatically determined. An example of this would be the use of logarithmic axis scaling. Though it is plausible that a generic enough visualization could be developed for most cases, it is likely that there will always be an edge case based on the usage scenario in which a user will have specific demands of the chart. In such cases, it will be required that the user modify the code in the visualization class itself such that their needs are met. Alternatively, a second modified version of the visualization could be created so that either could be generated alternatively in the same execution of an analysis program.

6.4 Usage

FROM THE USER PERSPECTIVE implementing a data flow which uses visualizations as described in this work is very simple. As described in Chapter 5 one must ensure that the visualizer and collector objects are instantiated, following which collection and visualization can be performed at will. A process diagram illustrating the interaction

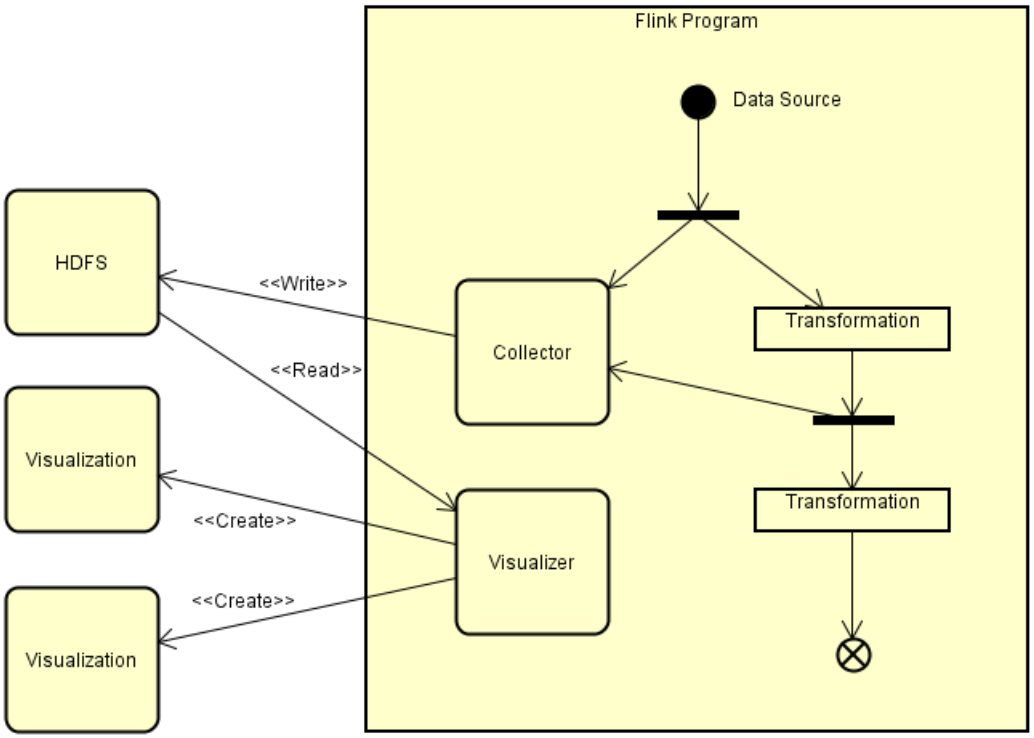

Figure 6.2: A diagram illustrating the visualization process

between the different objects involved and the order of operations can be seen in Figure 6.2.

Order of Events

During the collection phase, a data sink task is generated when the data from the job flow is written to the file system. After this task, an execution call is made on the environment to ensure that no further writes are made before the data is read into the visualizer independent of the original flow itself. Because the optimizer will generate plans in a way which is unpredictable to a user, there is no way of guaranteeing the order of operations which will occur before the visualization itself becomes available. However, because the visualization task runs without using Map-Reduce once the data is obtained in most cases it is performed quickly relative to the rest of the job flow. Additionally, because the data collection step necessitates an execution step, the following Map-Reduce operations are even more likely to be more expensive than the visualization itself.

Performance

In terms of added execution time incurred by the visualization process, we add only the time required to write data to the file system and to generate charts. In the case

of the write operation, this will of course be executed in linear time with efficiency being dependent on the extent to which the partitioning scheme is balanced. Once data has been provided for consumption by the visualizer, the remaining performance overhead depends on the efficiency of the visualization classes themselves. In the case of those visualizations and charts which have been identified as most important in this work, each was implemented in a way which required no operations with less that linear performance. This means that no standard visualization will be expected to take more execution time than a single data set transformation. In addition, generally the applications of most visualizations will operate on some summary or subset of the original data, meaning we can expect even faster execution. In cases where complex visualization classes (purpose built beyond the general cases described so far in this work) are introduced by a user, the efficiency of the user code will determine any effects on overall runtime. The strong performance of the visualization classes implemented for testing purposes relies on the assumption that certain features were not absolutely necessary for most in-situ analysis scenarios. For example, adding automatic sorting or outlier detection to visualization classes will significantly increase their algorithmic complexities.

CHAPTER 7

Future Work

7.1 Visualizations

THE VISUALIZATIONS USED SO FAR in this work demonstrate an adequate base level of functionality for achieving the desired results in most in-situ analysis scenarios. The obvious area for improvement here is in expanding the capabilities of the system to handle specific graphical tasks without the ad-hoc design of graphical classes. Because so much work has been done in recent decades in the field of formalizing visualization practices, some works have emerged which have had serious influences on most well-used libraries built for this task. One such work is "The Grammar of Graphics" by Leland Wilkinson [37].

Grammar of Graphics

The Grammar of Graphics is a seminal work in the field of scientific visualization which defines a rigorous method for developing graphics based on the data and scenario presented. This gives rigid reasoning behind the application of most common graphics in a scientific environment, and more tangibly has resulted in the creation of several graphics libraries which are capable of building any visualization described by the grammar. The book deals exclusively with static graphics as would be seen in a statistical or scientific analysis vs. some kind of interactive business visualization.

Libraries

Though there have been several projects and software libraries based on the theoretical foundations laid out in The Grammar of Graphics since its publication, there are no such libraries available in java. The reasons for this are complex, but generally speaking major factors include the widespread use of languages other than java for analysis and the complexity of implementation in java over other environments. Typically, when

scientific visualizations are needed analysts tend to turn to other languages in which a solution does exist.

R & ggplot

The statistical programming language R was purpose built with data analysis in mind. As such, it has a large user base across all scientific disciplines and has become a de facto standard for statistical computing and visualization. One of the most widely used libraries for visualization in R, is "ggplot" [38] which is based on "The Grammar of Graphics" and was first released only a year after the publication of the book. Ultimately, there is a strong argument for this library within R being the most convenient way of visualizing data by virtue of existing user base, documentation, and simple syntax alone. The difficulty of course is integrating R into the existing java code, though several approaches are available.

R in Java

There has been work in both directions of Java/R interoperability, with some R users desiring the performance or flexibility of specific java methods or libraries, and java users desiring the analytical tools provided by R. From R, the RJava project allows users to call java functions, and likewise the RCaller package allows java users to execute R code by making calls to a local script. The problem with such a method in the case of in-situ analytics lies in a reliance on the execution environment being configured for R. Additionally, with a library such as RCaller, there can be issues when executing in a cluster environment, as data transit has not been optimized for distribution beyond four nodes. A better solution in a project such as Renjin, which is a version of the R language written entirely in java, so that there are no requirements placed on the execution environment. For general purpose analytical R usage this is an elegant solution, but for visualization we rely on packages more than the R language itself, and the bulk of commonly used packages are written in C and have not yet been themselves rewritten to run under a JVM based environment such as a Renjin application.

Python

Python, though newer than R is quickly becoming as popular in the realm of data science and statistical computing in general. Though it doesn't yet have the same array of scientific computing libraries as R, there are several ongoing projects which aim to close the gap. Of course high on the list of items to be developed for python was a version of ggplot. This work has been largely completed, save for minor changes which are still being made based on user feedback and some small aesthetic adjustments which are ongoing. Like in R, there are projects which aim to allow for the execution of python code in java and vice versa. On particularly popular option among such projects is Jython, which like Renjin is an implementation of the python language written entirely in java. Unfortunately, the exact same pitfall is encountered here, with the modules required to

utilize the ggplot python library being built using C, and thus not usable in a JVM only environment.

JyNi

As opposed to the case with Renjin however, a compatibility layer has been developed for Jython with the express purpose of allowing the use of C based scientific python libraries in java applications. This compatibility layer is called "JyNi" [39] and is still actively in development. Unfortunately, this active development means that as of the time of this writing, ggplot is one of several libraries which still presents issues even with the compatibility layer. This issues is expected to be resolved by the end of 2015, and thus will soon be a viable option for accessing a very robust visualization library.

7.2 Real-Time Results

DISTRIBUTION IN ANALYSIS SYSTEMS following the general mapreduce model all operate very similarly in concept. This means that generally speaking, we can expect the dataset to be mapped into a set of key-value pairs which are then partitioned across a cluster in a uniformly distributed way. Because we may want to examine the intermediate dataset at a point prior to a reduce operation which would centralize the dataset, we must collect it piecemeal from each node in the cluster. This could be achieved by sending the datasets from each node in the cluster to the visualizer for summary.

Message Passing

Message passing could allow us to invoke a send message call from each in-situ data collector operating on a shard of the complete data set, and then receive it in the visualizer. The visualizer can perform whichever operations are needed in order to merge the datasets considering the original locations and timing in order to generate useful output. Many simple frameworks for message passing such as RabbitMQ are being used with major Apache projects and would be simple to include within this work without introducing additional dependencies or platform requirements. Additionally, as briefly mentioned in Chapter 4, some simple design patterns already exist for message passing in Map-Reduce jobs [25].

7.3 Interface

THOUGH IT DOES NOT PRESENT A RESEARCH PROBLEM, a major improvement to usability of this visualization method would be the inclusion of some kind of interactive interface. Because each of the visualized and collected datasets are indexed within the visualizer, they can be kept for offline access and organized for manipulation or orga-

nization by users. Such a user interface could appear in a similar fashion to the web interface provided for visualizing the flink execution plan. It would be simple to add interactivity and user input, which could potentially even be expanded to allow for the deeper analysis provided by som of the discussed methods such as phrase nets and word trees.

Javascript

While there are many limitations to the available visualization tools and libraries in Java, visualization is a much more common task in web based applications and thus visualization libraries built for the web are much more robust in general. Because the visualization classes in this work are mainly used for data formatting and are not particularly coupled to Java, the change to using a robust javascript library such as D3 over the processing tools used here would be simple.

7.4 Automation

CURRENTLY, CODE MODIFICATION IS REQUIRED for the data collector to be able to identify when to collect, and what format of data to expect. Additionally, the developer of an analysis program is expected to specify which type of visualization(s) to generate for each of the collected data sets. With the addition of some pattern matching and simple machine learning methods, it could be possible to perform these actions without the user needing to add any code to an analysis program.

Automatic Collection

Identifying points at which to collect data is a very simple problem. Any of the points of interest for collection will occur either immediately after a data set is populated or modified, or directly before the end of the program when the data is in its final state. In terms of the program execution plan, if each node in the generated data flow graph represents an transform or data source/destination, we can simply collect data at each edge. Complications could potentially arise in cases where control flow logic such as looping would generate an impractically large number of collected data sets with limited or useless semantic differences between one another; however, these situations could be mitigated by simple ignoring potential collection points from within complex control structures.

Automatic Visualization

Given the rigid nature in which the application of visualizations are often described, particularly in the case of the Grammar of Graphics [37], it is feasible that a method of determining the most appropriate visualization for a data set would be programmatically. The first major problem here would be in handling the relatively large number of potential input formats in which data may present itself in an analysis program and

detecting mismatches or errors therein. In this work, much of the complication is avoided by virtue of the developer-written calls to the collector, in which they are relied upon to provide accurate class information. Error detection and handling would have to be very robust in such a scenario, so that simple errors could be visualized for the user rather than simply causing failure in the visualization process. Once features such as type, dimensionality, and size of data have been detected through some mechanism, some conditional logic could be applied which would result in the program selecting the most appropriate general purpose visualization for the provided data set.

CHAPTER 8

Conclusion

APPENDIX A

A.1 Collection Code

THE FOLLOWING METHODS perform the important steps in collecting a data set from the executing job.

HDFS Collect Method

```
1   public void clusterCollect(int id, DataSet data, Class... c) throws Exception{
2
3       data.writeAsCsv(writeDir);
4       env.execute("Write");
5
6
7       //Read data back into new dataset
8       Configuration conf = new Configuration();
9       conf.addResource(new Path("/hadoop/projects/hadoop-1.0.4/conf/core-site.xml"));
10      conf.addResource(new Path("/hadoop/projects/hadoop-1.0.4/conf/hdfs-site.xml"));
11      FileSystem fs = FileSystem.get(conf);
12
13      File dir = new File(writeDir);
14      ArrayList<Tuple> dataSet = new ArrayList<>();
15      FSDataInputStream inputStream;
16      String line;
17
18      FileStatus[] status = fs.listStatus(new Path(writeDir));
19      for (int i=0;i<status.length;i++){
20          inputStream = fs.open(status[i].getPath());
21          while ((line = inputStream.readLine())!= null){
22              Tuple addedTuple = parseTuple(line, c);
23              dataSet.add(addedTuple);
24          }
25      }
26
27      visualizer.addData(new InSituDataSet(id, dataSet));
28      dir.deleteOnExit();
29  }
```

Local Collect Method

```
1   public void localCollect(int id, DataSet data, Class... c) throws Exception{
2       ArrayList<Tuple> dataSet = new ArrayList<>();
3
4       File outputDir = new File("CollectorWrite");
5       FileUtils.forceMkdir(outputDir);
6
7       //Write external data set to CSV
8       data.writeAsCsv(outputDir.getPath(), org.apache.flink.core.fs.FileSystem.WriteMode.OVERWRITE);
9       env.execute("Write");
10
11      //Read data originally from external data set into internal one
12      File dir = new File("CollectorWrite");
13      File[] directoryListing = dir.listFiles();
14      BufferedReader reader = null;
15      String line;
16
17      for (File file : directoryListing) {
18          try {
19              reader = new BufferedReader(new FileReader(file.getPath()));
20              while ((line = reader.readLine()) != null) {
21                  Tuple addedTuple = parseTuple(line, c);
22                  dataSet.add(addedTuple);
23              }
24          } catch (IOException e) {
25              e.printStackTrace();
26          } finally {
27              if (reader != null) {
28                  try {
29                      reader.close();
30                  } catch (IOException e) {
31                      e.printStackTrace();
32                  }
33              }
34          }
35      }
36      visualizer.addData(new InSituDataSet(id, dataSet));
37
38      try {
39          FileUtils.deleteDirectory(dir);
40      }
41      catch (IOException e) {
42          e.printStackTrace();
43      }
44  }
```

Bibliography

[1] N. Shoresh and B. Wong, "Points of view: Data exploration," *Nature Methods*, vol. 9, no. 1, p. 5, 2012.

[2] E. Tufte, "The Visual Display of Quantitative Information," *CT Graphics, Cheshire*, 2001.

[3] R. Ho, "Hadoop Map/Reduce Implementation," 2008.

[4] W. D. Pauw, M. Leţia, B. Gedik, and H. Andrade, "Visual debugging for stream processing applications," *Runtime Verification*, pp. 18–35, 2010.

[5] Apache Software Foundation, "Optimizer Plan Visualization Tool," 2014.

[6] S. Few, "Save the Pies for Dessert," *Visual Business Intelligence Newsletter*, pp. 1–14, 2007.

[7] G. Deleuze, P. Guattari, and F. Guattari, *A thousand plateaus: Capitalism and schizophrenia*, vol. 19. University of Minnesota Press, 1987.

[8] F. Van Ham, M. Wattenberg, and F. B. Viégas, "Mapping text with phrase nets," in *IEEE Transactions on Visualization and Computer Graphics*, vol. 15, pp. 1169–1176, 2009.

[9] M. Wattenberg and F. B. Viégas, "The word tree, an interactive visual concordance," in *IEEE Transactions on Visualization and Computer Graphics*, vol. 14, pp. 1221–1228, 2008.

[10] KONECT, "Konect network dataset."

[11] D. Battré, S. Ewen, F. Hueske, and O. Kao, "Nephele / PACTs : A Programming Model and Execution Framework for Web-Scale Analytical Processing Categories and Subject Descriptors," *ACM Symposium on Cloud Computing*, pp. 119–130, 2010.

[12] Y. Amsterdamer, S. B. Davidson, D. Deutch, T. Milo, J. Stoyanovich, and V. Tannen, "Putting Lipstick on Pig : Enabling Database-style Workflow Provenance," *Proceedings of the VLDB Endowment*, vol. 5, no. 4, pp. 346–357, 2011.

[13] B. . . . Gedik, H. . . . Andrade, K.-L. . . . Wu, P. S. . . . Yu, and M. . . . Doo, "SPADE: The system S declarative stream processing engine.," *Proceedings of the ACM SIGMOD*

International Conference on Management of Data, no. SIGMOD 2008: Proceedings of the ACM SIGMOD International Conference on Management of Data 2008, pp. 1123–1134, 2008.

[14] F. J. Anscombe, "Graphs in Statistical Analysis," *The American Statistician*, vol. 27, no. 1, pp. 17–21, 1973.

[15] S. Few, *Information Dashboard Design*. 2006.

[16] "A Structured Review of Information Visualization Success Measurement."

[17] S. Klasky, H. Abbasi, J. Logan, M. Parashar, K. Schwan, A. Shoshani, M. Wolf, A. Sean, I. Altintas, W. Bethel, C. Luis, C. Chang, J. Chen, H. Childs, J. Cummings, C. Docan, G. Eisenhauer, S. Ethier, R. Grout, S. Lakshminarasimhan, Z. Lin, Q. Liu, X. Ma, K. Moreland, V. Pascucci, N. Podhorszki, N. Samatova, W. Schroeder, R. Tchoua, Y. Tian, R. Vatsavai, J. Wu, W. Yu, and F. Zheng, "In Situ Data Processing for Extreme-Scale Computing," in *SciDAC Conference*, 2011.

[18] A. F. Gates, O. Natkovich, S. Chopra, P. Kamath, S. M. Narayanamurthy, C. Olston, R. Benjaminn, S. Srinavasan, and U. Srivastava, "Building a High-Level Dataflow System on top of Map-Reduce: The Pig Experience," *Vldb '09*, pp. 1–12, 2009.

[19] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins, "Pig Latin: A Not-So-Foreign Language for Data Processing," in *Proceedings of the 2008 ACM SIGMOD international conference on Management of data - SIGMOD '08*, p. 1099, 2008.

[20] N. Robbins, *Creating More Effective Graphs*. Hoboken: John Wiley & Sons, 2005.

[21] J. Kunegis, "KONECT - The Koblenz Network Collection," in *WWW 2013 Companion*, 2013.

[22] W. Didimo, G. Liotta, and S. Romeao, "Topology-Driven Force-Directed Algorithms," in *Graph Drawing 2010 Revised Selected Papers* (U. Brandes and S. Cornelsen, eds.), (Konstanz), pp. 165–176, Springer Berlin Heidelberg, 2011.

[23] D. Miner and A. Shook, *MapReduce Design Patterns*. Sebastopol: O'Reilly, first edit ed., 2012.

[24] M. Islam, A. K. Huang, M. Battisha, M. Chiang, S. Srinivasan, C. Peters, A. Neumann, and A. Abdelnur, "Oozie: towards a scalable workflow management system for Hadoop," in *Proceedings of the 1st ACM SIGMOD Workshop on Scalable Workflow Execution Engines and Technologies - SWEET '12*, pp. 1–10, ACM Press, 2012.

[25] J. Lin and M. Schatz, "Design Patterns for Efficient Graph Algorithms in MapReduce," *Mlg*, pp. 78–85, 2010.

[26] C. Blake and C. Merz, "UCI Repository of machine learning databases," 1998.

[27] R. Kohavi, "Scaling Up the Accuracy of Naive-Bayes Classifiers: a Decision-Tree Hybrid," in *KDD-96 Proceedings*, pp. 202–207, 1996.

[28] "Wikipedia conflict network dataset – KONECT," Oct. 2014.

[29] U. Brandes and J. Lerner, "Structural similarity: Spectral methods for relaxed blockmodeling," *J. Classification*, vol. 27, no. 3, pp. 279–306, 2010.

[30] R. Fisher, "The Use of Multiple Measurements in Taxonomic Problems," *Annals of Eugenics*, vol. 7, no. 2, pp. 179–188, 1936.

[31] B. Shneiderman, "Tree visualization with tree-maps: 2-d space-filling approach," 1992.

[32] B. B. Bederson, B. Shneiderman, and M. Wattenberg, "Ordered and quantum treemaps: Making effective use of 2D space to display hierarchies," *ACM Transactions on Graphics*, vol. 21, no. 4, pp. 833–854, 2002.

[33] T. Schreck, D. Keim, and F. Mansmann, "Regular TreeMap Layouts for Visual Analysis of Hierarchical Data," in *Spring Conference on Computer Graphics*, 2006.

[34] J. Wood and J. Dykes, "Spatially ordered treemaps," in *IEEE Transactions on Visualization and Computer Graphics*, vol. 14, pp. 1348–1355, 2008.

[35] A. Slingsby, J. Dykes, and J. Wood, "Configuring hierarchical layouts to address research questions," in *IEEE Transactions on Visualization and Computer Graphics*, vol. 15, pp. 977–984, 2009.

[36] J. D. G. G. Plaisant C Fekete, "Promoting Insight Based Evaluation of Visualization: From Contest to Benchmark Repository," *IEEE transaction on visualization and computer graphics*, vol. 14, no. May, pp. 120–134, 2008.

[37] L. Wilkinson, *The Grammar of Graphics*. Springer, 2nd ed., 2005.

[38] H. Wickham, "An Implemetation of the Grammar of Graphics in {R}: ggplot," *American Statistical Association 2006 Proceedings of the Section on Statistical Graphics*, pp. 1–8, 2006.

[39] S. Richthofer, "JyNI – Using native CPython-Extensions in Jython," no. Euroscipy, pp. 59–64, 2013.