Visualization of Data Flow Graphs for In Situ Data Analysis

Jacob Edwards

A thesis presented for the degree of
Master of Science

Database Systems and Information Management Group
Technische Universität Berlin
Berlin, Germany

31/07/2015

Author:
Jacob Edwards
Technische Universität Berlin
Berlin, 2015.

Abstract

TODO-> write abstract <-TODO

Acknowledgements

Contents

List of Figures

List of Abbreviations

DAG	directed acyclic graph
KPI	key performance indicator

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 [BEHK10], Pig [ADD$^+$11], or IBMs System S [**?**] to represent analysis jobs as a series of individual tasks. These tasks are connected into a data flow which generally takes the form of an directed acyclic graph (DAG), which provides a useful visual metaphor for the ordering and dependencies of each task within a job. While this is adequate for describing the process by which data is analyzed, it leaves much to be desired in terms of describing the data itself. In particular, in cases where execution times are particularly long. Thus far, few systems making use of data flow graphs have invested significantly in the area of visual feedback within these graphs. System S provides basic feedback indicating the status of dataset processing without real feedback regarding data features [PLGA10], and Lipstick has evolved from a method of providing provenance models for pig latin queries [ADD$^+$11] to providing rudimentary DAG visualization capabilities for Apache Pig in its current development state [ADD$^+$11].

1.2 Structure of this Thesis

Chapter 2	contains a survey of related work
Chapter 3	provides an overview of data types and models
Chapter 4	details the implementation
Chapter 5	results and conclusions

CHAPTER 2

Related Work

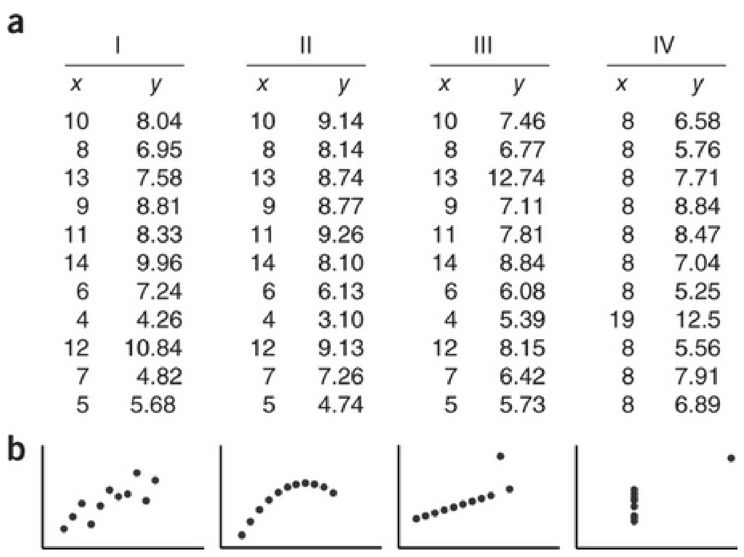
THE FIELD OF DATA VISUALIZATION has existed in some form for as long as data analysis has taken place. The primary purpose of data visualization is of course the effective communication of information through the use of graphics. Across varying fields and time periods, different approaches have been 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" [**?**]. 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"[Tuf83]. This work provided a summary of several different types of



I		II		III		IV	
x	y	x	y	x	y	x	y
10	8.04	10	9.14	10	7.46	8	6.58
8	6.95	8	8.14	8	6.77	8	5.76
13	7.58	13	8.74	13	12.74	8	7.71
9	8.81	9	8.77	9	7.11	8	8.84
11	8.33	11	9.26	11	7.81	8	8.47
14	9.96	14	8.10	14	8.84	8	7.04
6	7.24	6	6.13	6	6.08	8	5.25
4	4.26	4	3.10	4	5.39	19	12.5
12	10.84	12	9.13	12	8.15	8	5.56
7	4.82	7	7.26	7	6.42	8	7.91
5	5.68	5	4.74	5	5.73	8	6.89

Figure 2.1: Anscombe's Quartet [**?**]

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"[Tuf83]. 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.

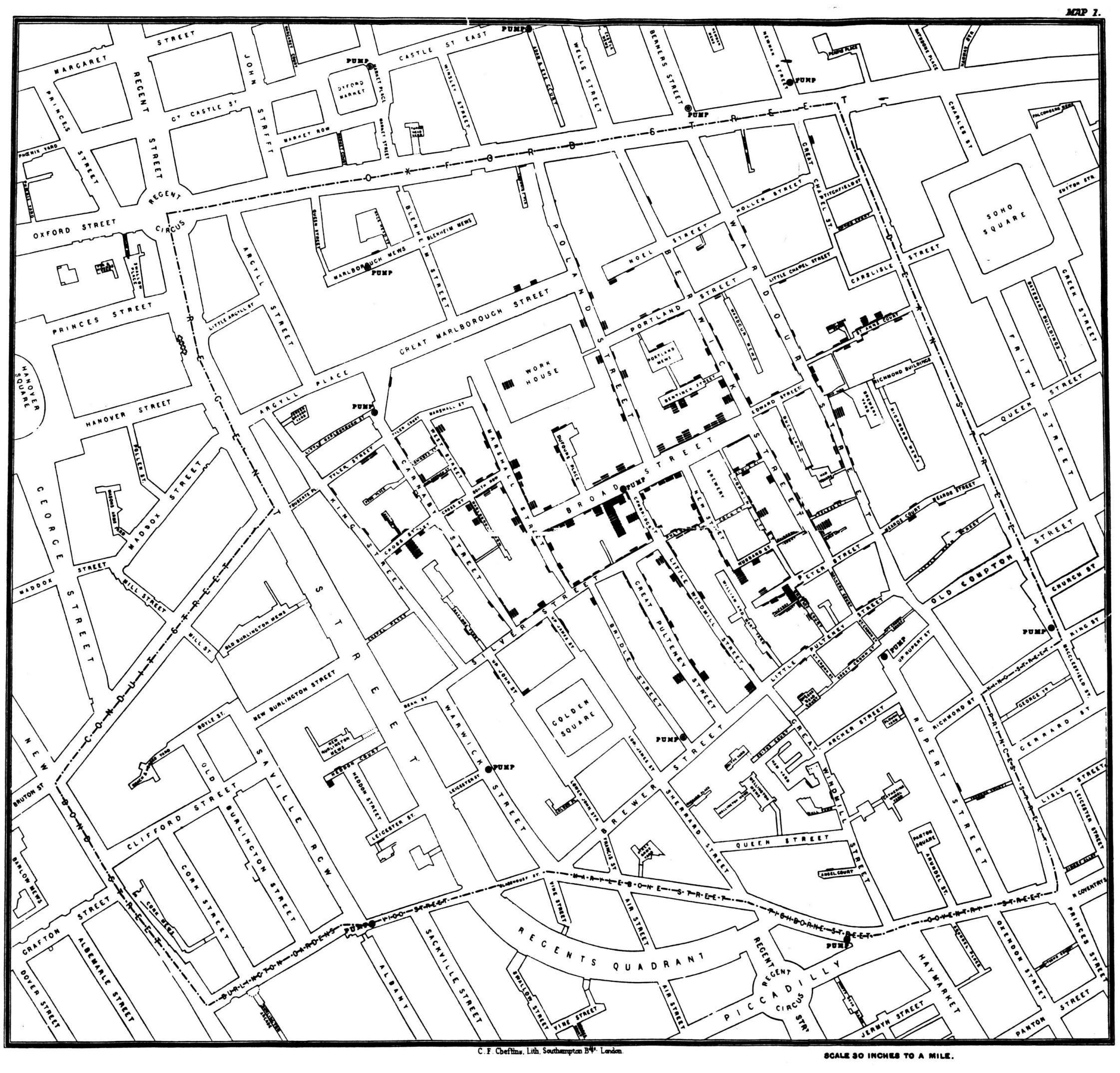


Figure 2.2: The map used by John Snow to determine the source of a cholera outbreak [?]

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" [Few06] 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.

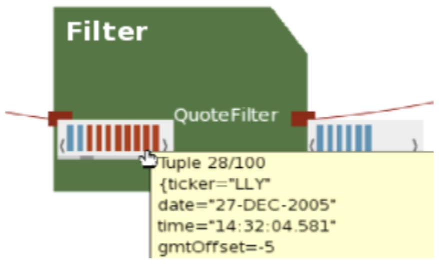
Evaluation of Visualizations

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 for handling such problems [KAL+11]. 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.

Pig

Flink



Tuple 28/100
{ticker="LLY"
date="27-DEC-2005"
time="14:32:04.581"
gmtOffset=-5

Figure 2.3: An executing operator as visualized in IBM's System S [PLGA10]

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 [**?**] and has included basic visualization of these graphs [PLGA10]. 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 2.3. These tuples can be highlighted in order to show specific data values, and to highlight data dependencies which exist downstream.

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 2.3 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 [ADD$^+$11], 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.

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.

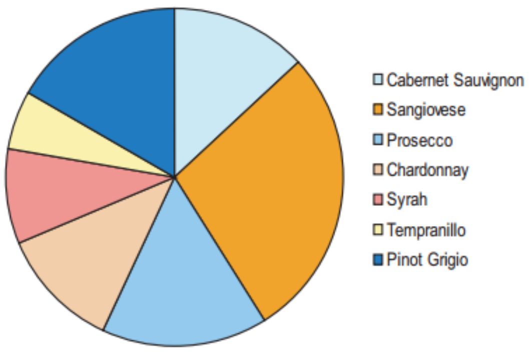


Figure 3.1: A pie chart showing proportions of wine varieties [**?**]

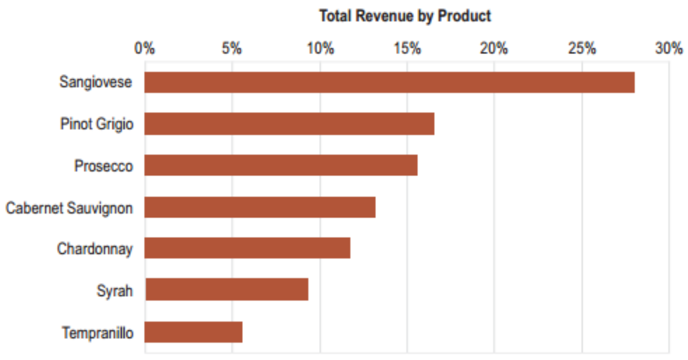


Figure 3.2: A bar chart showing proportions of wine varieties [**?**]

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 [**?**]. 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 3.1 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 [?], 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.

Word Trees [?]

Phrase Nets Phrase nets [?] 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 3.2 shows a phrase net

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

built using the old testament, which connects two words X and Y based on occurences of the phrase "X of Y" in the text.

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.

Graphs in DAG Systems
KONECT

The KONECT project [**?**] 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

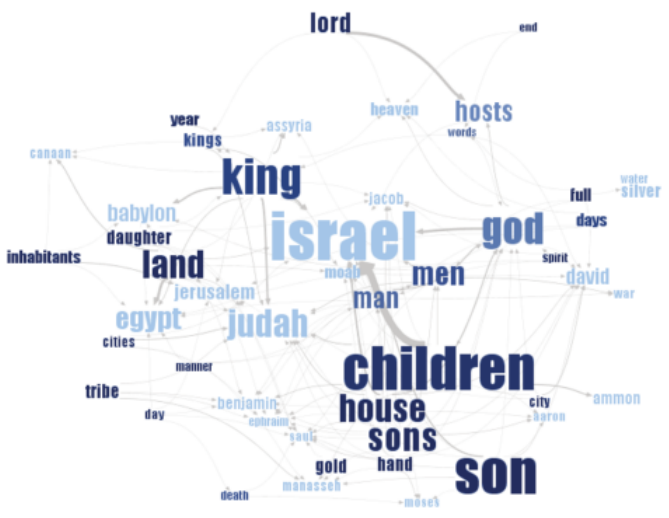


Figure 3.4: A phrase net visualizeing "X of Y" in the old testament [**?**]

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 3.3. 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 3.3. Of course, these distribution analyses can be focused on other 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

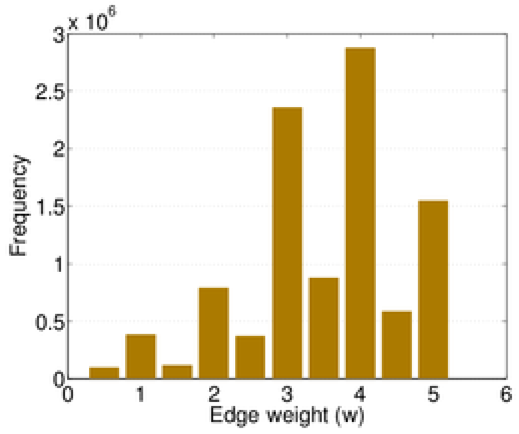


Figure 3.5: A visualization of edge weight distribution [?]

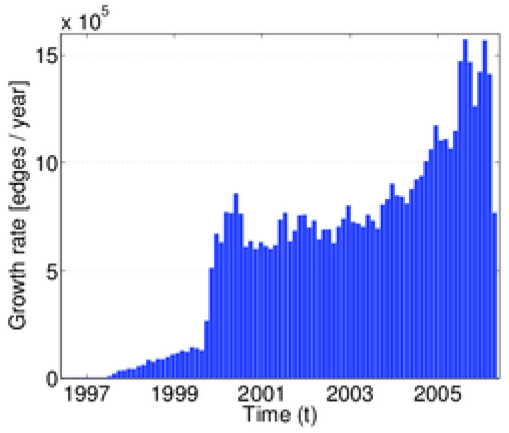


Figure 3.6: A temporal visualization of edge growth [?]

be seen in Figure 3.3. 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.

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

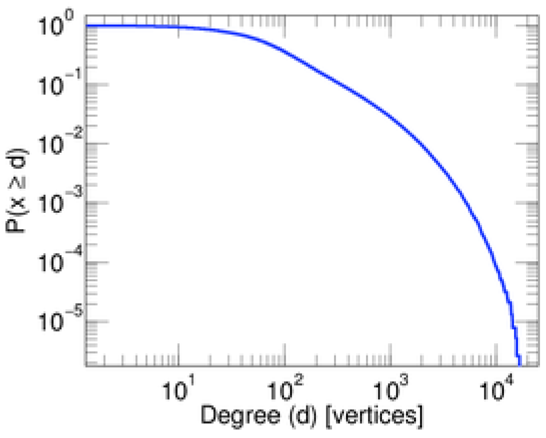


Figure 3.7: A visualization of cumulative distribution [?]

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 [?], 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

Usage Scenarios

THERE ARE MANY SCENARIOS in which MapReduce can be applied. Because this work is meant to applicable to any MapReduce job, tests have been selected in order to cover a varied range of analysis 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" [**?**]. In addition to this, the unique features of Flink are applied in order to establish that non-generic cases are also covered. Of course, in addition to the analysis itself the type of data being visualized is key. As such, these design patterns and features are applied to a varied array of data sources which necessitate the use of all of the most vital data visualization approaches.

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.

Shuffling

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.

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 [**?**]. 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

CHAPTER 5

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.

5.1 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 5.1 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 robust visualization libraries. In more complex and specific scenarios such as generating phrase nets, '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.

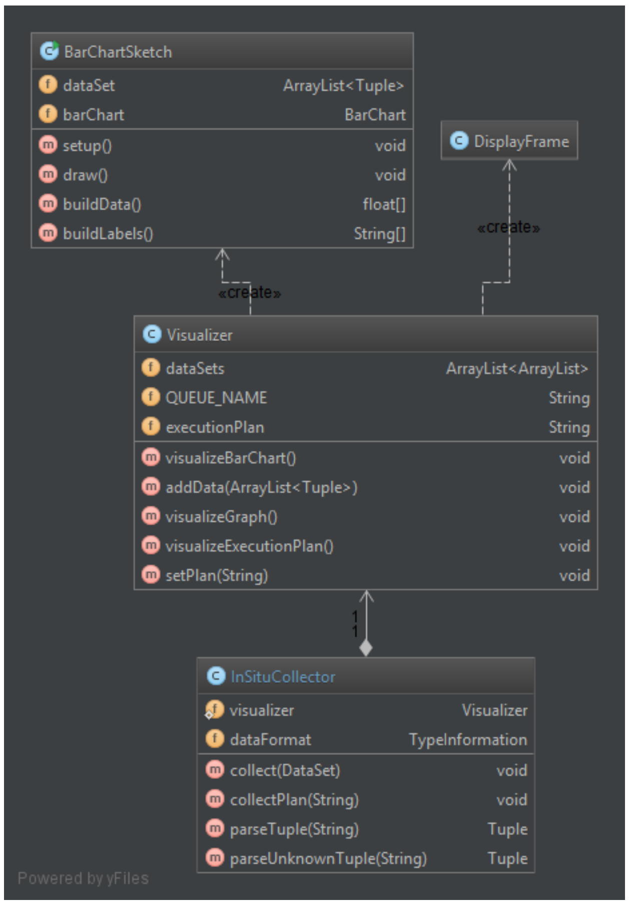


Figure 5.1: A UML diagram of the core classes

TODO-> Replace figure with updated/formatted version <-TODO

5.2 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.

Values

Values are types which have their serialization and de-serialization specified manually.

Hadoop Writeables

Objects which implement the Hadoop writeable interface.

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.

Collect Method

Each time the collect method is called, it sends a new dataset to the central visualizer class. This method accepts a dataset as its sole argument and writes this dataset to memory in a format which can be read by the data collector. The data collector then reads this data into a new dataset outside of the original analysis flow's execution environment.

Data Sets

A custom data set class exists for the use of the collector and visualizer. This class is very similar 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 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.

Type Erasure When analysis jobs are executed, the java compiler will erase types and operate exclusively with generics. This means that when this data is extracted, some additional work is needed in order to determine a sufficient approximation of the original type for storage in a custom data set. To handle this, as each record is read into a data collector they are parsed through a set of pattern matching checks which determine the number of fields and the fields' types. Firstly, a simple line split determines the size of the tuples which should exist in the data set based on the input record. Next, each field is checked individually using the java string utilities library to determine whether they are numeric or non-numeric. Fields in each of these categories are then passed through a cascading set of conditional checks which determine their specific basic type, from least to most complex. For example, this method will attempt to parse a numeric field as an integer, and upon failure attempt to parse the field as a long. This process continues until a match is found; in the case that one is not an exception is thrown.

5.3 Distribution

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 is achieved by sending the datasets from each node in the cluster to the visualizer for summary.

Message Passing Message passing allows us to invoke a send message call from each in-situ data collector operating on a shard of the complete data set, and then receiving 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.

Patterns? I'm not sure yet if a specific pattern will apply.

RabbitMQ So far arbitrary.

Specifics Implementation details such as server locale etc.

5.4 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 selecting and conceptualizing the most appropriate way to convey the information at hand. In building the visualizations in this work, several languages and libraries have been applied in order to complete the work in the most effective way possible.

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 the more complex 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.

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 classes from these utility libraries in the simplest of cases (such as the bar chart). In addition to providing basic visualizations in a pre-packaged format, there are some other tools such as navigational and formatting features which have been utilized in this work.

Swing

Outside of the visualizations themselves, the work of creating frames and navigation is largely handled through directly using java's swing visualization toolkit.

Presentation of Visualizations

More comprehensive packaging, eventually.

CHAPTER 6

Evaluation

6.1 Accuracy

6.2 Performance

6.3 Usefulness

APPENDIX A

Implementation

A.1 My Algorithm

THE FOLLOWING FUNCTION computes something

```
#include <cv.h>
using namespace cv;
// your code goes here
```

Bibliography

[ADD$^+$11] 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. [Online]. Available: http://dl.acm.org/citation.cfm?id=2095693

[BEHK10] 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. [Online]. Available: http://doi.acm.org/10.1145/1807128.1807148$\delimiter"026E30F$nhttp://dl.acm.org/citation.cfm?id=1807148

[Few06] S. Few, *Information Dashboard Design*, 2006. [Online]. Available: http://proquest.safaribooksonline.com/0596100167?suggested=top

[KAL$^+$11] 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. [Online]. Available: http://pasl.eng.auburn.edu/pubs/scidac11-adios-insitu.pdf

[PLGA10] W. D. Pauw, M. Leţia, B. Gedik, and H. Andrade, "Visual debugging for stream processing applications," *Runtime Verification*, pp. 18–35, 2010. [Online]. Available: http://link.springer.com/chapter/10.1007/978-3-642-16612-9_3

[Tuf83] E. Tufle, "The visual display of quantitative information," *CT Graphics, Cheshire*, 1983. [Online]. Available: http://www.colorado.edu/UCB/AcademicAffairs/ArtsSciences/geography/foote/maps/assign/reading/TufteCoversheet.pdf

Declaration of Authorship

I declare that the work presented here is, to the best of my knowledge and belief, original and the result of my own investigations, except as acknowledged, and has not been submitted, either in part or whole, for a degree at this or any other university.

Formulations and ideas taken from other sources are cited as such. This work has not been published.

Berlin, 31 July 2015

Jacob A. Edwards