Analysis and Improvement of Data Handling Performance of a Visualization Tool

by

Dennis Sogeler

Supervisors:

Prof. dr. Paul de Bra (TU/e)
Dr. Erik-Jan van der Linden (MagnaView B.V.)

Eindhoven, August 2006
Abstract

In the field of data analysis a gap can be observed between data handling tools on one hand, and visualization tools on the other hand. Data tools do not offer the best visualizations, whereas visualization tools lack performance when dealing with large amounts of data. Visualizations can help in data analysis. Therefore, the capabilities of data tools and visualization tools should be combined.

MagnaView is a visualization tool, which uses treemaps to create visualizations from data. MagnaView has evolved to an application with good visualization capabilities and many options to help create visualizations. However, in the development, the performance of data handling has stayed behind. Especially, when handling large amounts of data, it lacks performance.

In this thesis, we analyze the performance of MagnaView. We identify the bottlenecks in the application and present a solution to the bottlenecks that are related to the data handling.
# Table of Contents

**ABSTRACT** ............................................................................................................................. 3  
**TABLE OF CONTENTS** ........................................................................................................... 4  
0. **Preface** ................................................................................................................................. 6  
1. **Introduction** ......................................................................................................................... 7  
   1.1. Introduction .................................................................................................................. 7  
   1.2. Overview ..................................................................................................................... 8  
2. **Visualization tools** ............................................................................................................... 9  
   2.1. Introduction ................................................................................................................ 9  
   2.2. Treemap ....................................................................................................................... 10  
   2.3. Microsoft Research Treemapper ............................................................................. 11  
   2.4. Panopticon Explorer .NET ........................................................................................ 13  
   2.5. Microsoft Excel .......................................................................................................... 14  
   2.6. MagnaView ................................................................................................................ 15  
   2.7. Conclusion ................................................................................................................... 16  
3. **MagnaView** .......................................................................................................................... 18  
   3.1. Introduction ................................................................................................................. 18  
   3.2. Architecture ............................................................................................................... 19  
   3.3. Measurements ............................................................................................................ 20  
      3.3.1. User behaviour .................................................................................................. 21  
      3.3.2. MagnaView Benchmark ................................................................................. 21  
      3.3.3. AQTime ........................................................................................................... 21  
      3.3.4. Measurements ................................................................................................. 21  
   3.4. Bottlenecks ................................................................................................................... 22  
4. **Data handling** ...................................................................................................................... 25  
   4.1. Introduction ................................................................................................................. 25  
   4.2. Formalization .............................................................................................................. 25  
   4.3. Unique values ............................................................................................................. 26  
   4.4. Data sets ...................................................................................................................... 28  
      4.4.1. Theoretical data sets ....................................................................................... 29  
      4.4.2. Real world data sets ......................................................................................... 29  
5. **Data loading** ....................................................................................................................... 31  
   5.1. Introduction ................................................................................................................. 31  
      5.1.1. Number of columns ......................................................................................... 32  
      5.1.2. Number of rows ............................................................................................... 33  
      5.1.3. Number of unique values ................................................................................. 34  
   5.2. Data loading overview ................................................................................................. 35  
   5.3. Data loading algorithm ............................................................................................... 36  
   5.4. Complexity ................................................................................................................... 37  
   5.5. Bottlenecks ................................................................................................................... 38
0. Preface

This master’s thesis is part of the Computer Science program of the author at the Technische Universiteit Eindhoven.

In the early spring of 2005 I got in touch with MagnaView, when I was looking for an internship. Together with Erik-Jan van der Linden and Roel Vliegen, an interesting assignment for an internship in the area of visualization was formulated. After successfully finishing my internship at MagnaView, I had enjoyed my time at MagnaView so much, we discussed the possibilities for working on my master’s thesis at MagnaView.

For my master’s thesis I was looking for a challenging subject, which would join in with my interests in information systems and databases. It turned out that MagnaView was able to offer interesting subjects. Data handling in MagnaView became the subject for my master’s thesis. This subject was not only very challenging, but also would give me the opportunity to make a useful contribution to MagnaView.

I would like to thank all people who have helped and supported me in creating this thesis. First of all, I would like to thank Erik-Jan van der Linden for offering me all the possibilities I have had at MagnaView. I am also thankful to all my colleagues for creating a very pleasant and supportive atmosphere. Special words of thanks go to Roel, who was always there to help and gave many valuable comments.

I also want to express my gratitude to my supervisors Paul de Bra and Erik-Jan van der Linden for their motivation and ideas.

Special thanks go out to my parents, my sister and my girlfriend, who have always supported me in my education.

Dennis Sogeler
August 2006
1. Introduction

1.1. Introduction
Nowadays, a lot of data is collected, processed and stored in corporate environments. The value of this data for an organization depends, amongst others, on the possibilities to extract information from this data, which can be used to control and change the way the organization operates. It is, however, difficult to retrieve useful information out of the bits and bytes, and even if the information is retrieved, it is hard to interpret. Data and information extracted out of data can be understood faster if the data is visualized in some way; humans are able to easily process visual information.

The area of data visualization combines the scientific fields of data handling and visualization. A lot of research has been carried out in both of these fields, leading to insights in how to handle large amounts of data and in what good visualizations are. Based on these insights many applications have been and are still being developed, that all try to solve the data interpretation problem. In general, we can divide the applications that operate on the border of data handling and visualization in two groups: data tools with limited visualization capabilities and visualization tools with limited data handling capabilities.

Data tools try to solve the problem of handling large amounts of data. These tools provide ways to add, delete and modify data. Operations to search the data are also available. Data tools focus on how data is handled and not what the data represents. The second group of applications are visualization tools. These tools provide ways to convert data into visualizations. They focus on what the data represents, how the data is handled is less important as we will show.

There is a boundary between the two groups of applications. In general, data tools are not capable of visualizing what the data they manage represents. However, visualization tools focus on this point, but they only succeed in this when the amount of data is small. First, large amounts of data require special visualization methods. Second, the applications are not designed to handle large amounts of data.

A visualization tool that tries to bridge the gap between data tools and visualization tools is MagnaView. The goal of this application is to give an almost unlimited number of visualizations of business data. It uses a special visualization method, called treemaps. Some visualization methods are good when a global overview of the data is needed. Other visualization methods are suited when specific details are required. Treemaps provide a middle ground between these two by trying to combine both. Treemaps are also suited for visualizing large amounts of data.

MagnaView was originally developed as part of a Master’s Thesis at the Visualization Department of the Technische Universiteit Eindhoven. The emphasis in the development
of this application was on the visualization aspect and not on the data handling. The data handling part in the program is working, but far from optimal.

In this thesis we use MagnaView as the basis for creating a visualization tool that is at the same time capable of handling large amounts of data. The goal is to create a visualization tool with good data handling performance, that is, to bridge the gap between data tools with limited visualization capabilities and visualization tools with limited data handling capabilities. Henceforth, the central question of this thesis is:

\textit{Is it possible to improve data handling capabilities in a visualization tool to reduce the gap between data tools and visualization tools?}

In this thesis, we give an answer to the question. We present a number of improvements on the data handling part of MagnaView in particular and for treemap visualization applications in general. In order to focus our research, we set out to achieve a tenfold speedup of MagnaView.

1.2. Overview

Chapter 2 of this thesis presents an overview and comparison of a number of visualization tools. In Chapter 3 MagnaView is introduced. We discuss the architecture of MagnaView and determine the bottlenecks in this application. Chapter 4 discusses some aspects of data handling. In Chapter 5, we focus on the data loading in MagnaView. We give an overview of the data loading algorithm and identify the bottlenecks. Chapter 6 presents a new data loading algorithm, which gives a solution to the bottlenecks. In Chapter 7, we discuss building trees. First, an overview is given of the old method for building trees. Next, a new algorithm is presented, which improves the performance. In Chapter 8 an overview of the performance improvements is given. Finally, in Chapter 9 we present some conclusions and future work.
2. Visualization tools

In this chapter we take a look at some applications that are able to visualize business data. We focus on applications that use treemaps to visualize data and do not consider general Business Intelligence tools. In this chapter we also consider data tools with visualization capabilities. We focus on the data handling part of the applications and in particular the ability to visualize large amounts of data.

2.1. Introduction

There is a number of applications that visualize data using treemaps. For some of them the data to be visualized is pre-determined (e.g. the contents of a hard drive [9]), but we are more interested in applications that let the user visualize their own data. The best-known application to this respect is Treemap 4.1 [5]. It is developed at the Human-Computer Interaction Lab (HCIL) at the University of Maryland by Shneiderman, who is also the inventor of the concept of treemaps [2].

Some other applications we discuss in more detail, include the following: Microsoft Research Treemapper, Panopticon Explorer .NET and MagnaView. Finally, we consider the data handling and visualization capabilities of Microsoft Excel. This application is not able to produce treemap visualizations, but since this program is used by a large number of people to visualize (business) data in general, we use it in our comparison. This application is a good example of a data tool with visualization capabilities.

We try to compare these applications on a number of characteristics. These characteristics, together with a short description, are listed below.

- **Performance**
  This characteristic specifies the speed of the program. This is determined mainly by the loading time of the data and the time it takes to create a visualization out of it. The performance cannot always be measured in detail, but the resulting value for this characteristic is as objective as possible. When no detailed timings, such as log-files are possible, we measure the time by hand and look at the time taken by the processor of the computer. To make a fair comparison, we use the same data set for each application to measure the performance. This data set, which we will call Shop2, consists of 48,819 rows and 22 attributes. We have chosen this data set, because this is the largest data set which all applications can load successfully. For applications that could handle larger data sets, we also have used these, to get a good overview of the performance in handling large amount of data. We only give few characteristics of the data sets; these are discussed in more detail in section 4.4.2.

- **Visualization methods**
  This characteristic specifies how the data can be visualized. For this characteristic, the different visualization methods and their options are taken into account. Also the number of different methods and the scalability for large data sets is taken into account.
Hierarchical structure
This characteristic specifies to what extent data can be structured in a hierarchical way. Some visualizations require a hierarchical structure, whereas this is optional for other methods. The ease of creating and rearranging the hierarchical structure is also taken into account.

Flexibility
This characteristic represents the number of options in the application and the ease of use for the application.

2.2. Treemap
Treemap is one of the best known treemap visualization applications. It is one of many versions developed at the Human-Computer Interaction Lab (HCIL) at the University of Maryland. Founding director of the HCIL is Ben Shneiderman, who also came up with the concept of treemaps in the early 1990s [2]. At the HCIL a number of treemap applications are developed, the most recent (2003) is Treemap 4. We used version 4.1.1. for our comparison [5]. A screenshot of Treemap can be seen in Figure 1.

![Screenshot of Treemap 4.1](Figure 1 - Screenshot of Treemap 4.1)

We now consider the characteristics listed in the introduction.

Performance
In this application loading the data and creating a visualization out of it, is a one way process. Loading and visualizing data set Shop2 takes 7 seconds. The same process for a data set, which we will call Education1, consisting of almost 200,000 rows and 20 attributes, takes 33 seconds. After the loading of this amount
of data, the performance of the program is poor; actions that deal with the hierarchical structure take up to ten seconds. This application is not developed for large data sets. It standard uses a limited amount of memory, which is sufficient for most small data sets. It is however possible to visualize larger data sets, but only if the maximum amount of memory allocated by the application is increased manually. Although this thesis does not focus on memory requirements, the amount of memory required by this application is exceptional compared to all other application in this overview; for the Education1 dataset Treemap allocates over 450 Mb.

- **Visualization methods**
  This application offers only one visualization method, namely a treemap. There are many options that can be adjusted for the treemap visualization, such as partitioning method, size and colour of the borders and the nodes. The scalability of treemaps is large: they can be used for small data sets as well as large data sets.

- **Hierarchical structure**
  This application uses hierarchical structures, since a treemap is an image of hierarchical data. It is easy to specify any hierarchical structure; this can be done in the Graphical User Interface. It is not possible to specify options for the various levels of the hierarchical structure.

- **Flexibility**
  The level of flexibility of this application is high. There are many options that can be adjusted, and it is possible to create many different visualizations out of a data set. However, the formatting of the data set must meet some criteria, which restricts the flexibility.

Overall, this application provides many options for treemap visualizations. This application however is not suited to handle large data sets. From the documentation of Treemap: “We have not focused on optimizing Treemap for very large data files (...). Nevertheless we can say that Treemap was successfully used in the following case: Data Files with 25,000 nodes along with 8-9 attributes ” [5]. This confirms our conclusion, since these numbers are not even close to the amounts of nodes and attributes in large data sets.

### 2.3. Microsoft Research Treemapper

As its name suggests, Microsoft Research Treemapper is not a commercial application, but a research project from Microsoft. Therefore, its functionality is limited, but sufficient to create treemaps from data. It offers a very simple Graphical User Interface, from which data files can be loaded and some settings can be adjusted. We used version 1.0.1.19 for our comparison [6]. A screenshot of Microsoft Research Treemapper can be seen in Figure 2.
We now consider the characteristics listed in the introduction.

- **Performance**
  In this application loading the data and creating a visualization out of it, is a one way process. Loading data set *Shop2* and visualizing it takes 51 seconds. After the initial loading step, the application has a good performance, but this also is due to the limited flexibility.

- **Visualization methods**
  This application offers only one visualization method, namely a treemap. Except for the background colour, the colour and size of the borders, there are no options that can be adjusted for the treemap visualization. In the treemap it is possible to see which information is visualized where, using labels and tooltips. It is also possible to zoom into the treemap, to inspect parts of it in detail.

- **Hierarchical structure**
  This application uses a hierarchical structure, since a treemap is an image of hierarchical data. It is possible to specify any hierarchical structure, but this has to be done in the data file. This requires the data file to be in a specific format, where each relation is specified.

- **Flexibility**
  The level of flexibility of this application is very low. There are only few options to adjust and the hierarchical structure of the data has to be specified in the data file, which means that for every new hierarchical structure a new data file has to be created.

Overall this application offers only very basic treemap visualizations. It offers only few parameters that can be adjusted in the visualization and the biggest problem is that this application requires data files to be in a specified format.
2.4.  **Panopticon Explorer .NET**

Panopticon Explorer .NET is a visual data analysis application, created by Panopticon. It uses treemaps to visualize data. This application is one of the few commercial treemap applications. We used version 1.1.0.20959 for our comparison [7]. A screenshot of Panopticon Explorer .NET is presented in Figure 3.

![Figure 3 - Screenshot of Panopticon Explorer .NET](image)

We now consider the characteristics listed in the introduction.

- **Performance**
  This application offers a good performance. Loading data and creating a treemap is a one-way operation. For data set Shop2 this process takes about 13 seconds. During this process, the application creates multiple (although very simple) hierarchies out of the data. The performance of the application is good, most of the settings are applied real-time.

- **Visualization methods**
  This application offers only one visualization method, namely a treemap. There are some options that can be adjusted for the treemap visualization, such as colours, labels and tooltips.

- **Hierarchical structure**
  This application uses hierarchical structures, since a treemap is an image of hierarchical data. Initially, the application creates a number of very simple hierarchical structures. These structures can be edited in the Graphical User
Interface, although not in a very intuitive way. It is not possible to specify options for the various levels of the hierarchical structure separately.

- *Flexibility*
  The application has a large number of options that all can be adjusted. Filtering on data is possible, as well as zooming. The GUI accommodates too many options, leading to a clutter of all possible settings. The application can load a number of different file types. A small drawback is the unsupported txt file. According to the documentation, loading data from databases also is an option, although this is not supported in the evaluation version we used.

Overall, Panopticon Explorer .NET can handle data sets with many items and the performance after the loading process is good. The GUI, however, is not very intuitive, thus degrading the usability of the application.

### 2.5. Microsoft Excel

Microsoft Excel is one of the most used spreadsheet programs. This application is part of the Microsoft Office Suite. The current version is the 11th version for the Windows platform already. The first version appeared over 20 years ago. We use Microsoft Excel here as the prime example of a data tool with limited visualization capabilities. We used version 11.6560.6568 SP2 of Microsoft Excel 2003 for our comparison [8]. A screenshot of Microsoft Excel is presented in Figure 4.

![Figure 4 - Screenshot of Microsoft Excel](image)

We now consider the characteristics listed in the introduction.

- *Performance*
  The performance of this application is good, especially the data handling. Loading data set Shop2 takes only 2.5 seconds. Since Excel is a data tool, which has had
over 20 years of development, this result is of no surprise. The application lacks
the possibility to load large data sets however; the maximum number of rows in a
dataset is $65,536 (2^{16})$.

- **Visualization methods**
  Various types of visualizations are offered; most of which are standard data
  visualization types, such as pie charts and (stacked) bar charts. Unfortunately
  there is no option available to visualize data using treemaps. The existing methods
  for visualization are not well suited to deal with large amounts of data. Typically
  only very few data points, such as aggregates, can be used to keep the overview in
  the visualization.

- **Hierarchical structure**
  This application does not use any hierarchical structure.

- **Flexibility**
  There are many options for editing and converting data. The number of options
  for visualizations is smaller, although most important settings can be adjusted.
  Among these are type, axis and colour.

Overall, Excel is a typical data tool. The performance for the data handling part is very
good, but this application is suited for visualizing large data sets. It offers many
visualization methods, but their scalability for large amounts of data is not very good.

### 2.6. MagnaView

The final application we consider is MagnaView. This is a relative new commercial
application, which uses treemaps as the main visualization method. We use version 2.0
for our comparison [1]. A screenshot of MagnaView is presented in Figure 5.

![Figure 5 - Screenshot of MagnaView](image-url)
We now consider the characteristics listed in the introduction.

- **Performance**
  The performance of this application is not very good. Loading a data set and creating a visualization can be done in two separate steps. Loading data set Shop2 takes 19 seconds, creating a treemap out of it can take up to 3 seconds. The latter is done every time any setting of the treemap is altered, which is very often.

- **Visualization methods**
  This application uses treemaps as the main visualization method. It is possible, however, to simulate other visualization methods by applying transformations to treemaps; MagnaView tries to combine the advantages of treemaps with other visualization methods, like pie charts and (stacked) bar charts. This can be seen in Figure 5, where a treemap is presented with the appearance of a stacked bar chart.

- **Hierarchical structure**
  This application uses hierarchical structure, since a treemap is an image of hierarchical data. It is very simple to create and rearrange the various different levels of the hierarchical structure using the GUI. For each separate level of the hierarchical structure, options can be set.

- **Flexibility**
  On this point the application performs very well. First of all, there are many formats supported for data files. Next, there are many settings that can be adjusted for the treemap. Next to the options available in the other treemap applications, MagnaView distinguishes itself by offering all those settings per level of the hierarchical structure.

Overall, this application provides good visualizations, with an enormous flexibility. Especially the combination between treemaps and other visualization methods is special. The possibilities for visualizing data with this application are almost unlimited, although it lacks performance.

### 2.7. Conclusion

For a number of visualization tools and one data tool, we have discussed a number of characteristics. To provide a comparison, we give an overview of the tools. We present per application for each of the characteristics an (subjective) assessment, varying from “++”, indicating that the application performs well for this characteristic, to “−−”, indicating that the application performs poor for this characteristic. The results are presented in Table 1.

<table>
<thead>
<tr>
<th>Application</th>
<th>Performance</th>
<th>Visualization methods</th>
<th>Hierarchical structure</th>
<th>Flexibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>Treemap</td>
<td>−</td>
<td>o</td>
<td>+</td>
<td>0</td>
</tr>
<tr>
<td>Research Treemapper</td>
<td>−−</td>
<td>−</td>
<td>o</td>
<td>−−</td>
</tr>
<tr>
<td>Panopticon Explorer</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>0</td>
</tr>
<tr>
<td>Microsoft Excel</td>
<td>++</td>
<td>o</td>
<td>−</td>
<td>+</td>
</tr>
<tr>
<td>MagnaView</td>
<td>−</td>
<td>++</td>
<td>+</td>
<td>++</td>
</tr>
</tbody>
</table>

*Table 1 - Comparison of visualization tools*
We can make a distinction between data tools with visualization capabilities and visualization tools with data handling capabilities. There is one data tool in our comparison, Microsoft Excel, the others are visualization tools. In general, the data handling performance of visualization tools is not very good. These tools, however, offer possibilities for visualizing large amounts of data, especially by using treemaps. MagnaView is the most versatile visualization tool of all applications in our comparison. It offers good visualization methods and an enormous flexibility. Potentially this application can be used to visualize large data sets, it only lacks the performance for doing so.

Therefore, in order to create a tool with both adequate visualization capabilities, and good data handling performance, we use MagnaView as a starting point, and improve its data handling.
3. MagnaView

In this chapter we give an overview of MagnaView. First, we discuss the features of MagnaView. Next, we discuss the architecture of the application in more detail. In the last part of the chapter we identify a number of bottlenecks in the performance of MagnaView.

3.1. Introduction

MagnaView is a successor of the well-known application SequoiaView [9], developed at the Technische Universiteit Eindhoven. SequoiaView enables users to visualize the contents of hard drive(s). SequoiaView uses cushioned [4], squarified treemaps [3] for this purpose. MagnaView uses these same techniques (among others) for the graphical representation of data. MagnaView is a more generalized version of SequoiaView, i.e., MagnaView can handle all kinds of data, whereas SequoiaView is restricted to representing the contents of hard drives. MagnaView also provides more options and possibilities for configuration to users.

MagnaView can load (business) data from various sources and transform this data into tree structures and visualize it using treemaps. MagnaView can combine treemaps with business graphics.

We give an example of the treemaps MagnaView can create. We use data set Education1 from an educational institution. The data contains grades for students. For each grade there are various attributes available, such as teacher, course and student information. We can use this data set to create multiple views with different underlying hierarchies. Variations in the hierarchical structure can lead to different insights in the same data set.

Figure 6 gives an overview of the full data set. There is no hierarchical structure specified. The colours in the treemap are associated with the values of the grades. From this treemap it is possible to get an overview of the number of items in the data set.

Figure 6 - Treemap without hierarchical structure

In the treemap of Figure 7 a hierarchical structure is specified. The grades are sorted by their value, varying from grades from 9 to 10 in the yellow group at the top of the treemap to the red group of grades marked as “insufficient”, in the bottom of the treemap. In this data set, almost all grades are sufficient.

Figure 7 - Treemap with one level in the hierarchical structure: grade
In the treemap of Figure 8 one further level in the hierarchical structure is added. First, the grades are divided into the several different courses. Next, the grades are sorted by their value again. From this treemap an overview of the grades for each course can be obtained: some courses result in lower grades than others. This treemap is formatted to represent a stacked bar chart.

Figure 8 - Treemap with two levels in the hierarchical structure: course, grade

In the treemap of Figure 9 another level is added to the hierarchical structure. First, the grades are divided by the school they belong to. Next, the division is equal to the treemap presented in Figure 8. This visualization gives an overview of the number of grades for each different school (size of the rectangle on the highest level in the hierarchical structure), but also gives an overview of the results per course per school.

Figure 9 - Treemap with three levels in the hierarchical structure: schoolname, course, grade

In the last treemap, Figure 10, the results are divided per teacher. The results are presented in a pie chart, which is divided into the several groups of values of the grades. This treemap presents an overview of the grades given to the students for every teacher; it shows which teachers give lower or higher grades.

Figure 10 - Treemap with two levels in the hierarchical structure: teacher, grade

Next to these examples, there are many more visualizations one could think of, such as grades per gender, grades per student and grades per class.

### 3.2. Architecture

We now present an overview of the architecture of MagnaView. The model shown in Figure 11 gives an overview of MagnaView and the environment and the way users interact with MagnaView.
Users interact with the Graphical User Interface (GUI) of MagnaView, where they can define what data should be visualized and how it should be visualized. These definitions together are called the views. The input for MagnaView is business data. A tree structure is generated from the data, based on the settings specified in the view. Based on this tree structure and other settings from the view a treemap is generated and presented to the user in the GUI. The user has the possibility to explore this treemap interactively.

MagnaView is designed to extend the use of treemaps to business graphics. Originally, it was a research project, which later became a commercial application. The application has good visualization methods, but from the user’s perspective the performance is poor.

In this succeeding part of the chapter we present an overview of the bottlenecks in the performance of MagnaView. First, we give an overview how these bottlenecks were determined. Next, we give a summary of the bottlenecks.

3.3. Measurements

MagnaView has many options and possibilities, but in this thesis we concentrate on the data handling part. We give an overview of all the bottlenecks in the performance, as complete as possible, but we only try to find a solution for the bottlenecks related to the data handling part.

We used three different methods to determine the bottlenecks. Each of them has its own (dis)advantages, but the combination of the three is a good way to determine the bottlenecks in the application. We discuss each method separately.
3.3.1. User behaviour
First of all, we tried to identify bottlenecks by acting like a regular user. By using the application, one can get a feeling about the performance of the program. This method is not precise at all, but it quickly provides a rough idea of the performance. This method measures the performance from the user’s point of view and thus provides subjective measurements that are not necessarily the same as the actual objective measurements. Some actions can feel slow, while others that may be equally slow do not feel that way.

3.3.2. MagnaView Benchmark
The second method we used is an application we developed to measure the performance of each separate part of the program. This application is called MagnaView Benchmark. It imitates the user’s behaviour automatically and measures the time for each action. A screenshot can be seen in Figure 12.

It is possible to perform multiple benchmarks at once. The results can be saved to file automatically. This application provides an accurate method to benchmark the performance and show the timings of each separate action. Also performance improvements can be benchmarked against earlier versions. The performance of the program is only slightly influenced by the benchmark, depending on the amount of separate actions to be measured.

3.3.3. AQTime
The last method we used is a so-called profiler (AQTime, version 4.81.612.0 [10]). A profiler measures the performance of each separate line of code, and therefore provides a very detailed insight in the performance of an application. The measurements may not represent the real world timings, because the measurements influence the performance greatly. The process of measuring may consume more time than the actual code to be profiled. The profiler does however provide useful statistics about the code, such as the number of times each function is called and how functions depend on each other. Although this method provides detailed results, it is cumbersome and time consuming. Therefore, we only used the profiler to obtain information about the statistics mentioned.

3.3.4. Measurements
All timings displayed in this thesis are based on measurements from MagnaView Benchmark. They are based on series of measurements (at least 10), the average of which was calculated. Wherever timings are used to compare an action using different versions of the application, the measurements are run under equal conditions.
All measurements are performed on a personal computer, with the following specifications:

- **CPU:** Intel Pentium 4, 3.40 Ghz
- **Internal Memory:** 1024 Mb
- **Operation system:** Windows XP64

### 3.4. Bottlenecks

The approach chosen is to identify a list of actions in the applications that contribute most to the poor performance from the user’s perspective. From this list the actions that are related to the data handling part are chosen to be optimized. Other actions could be optimized as well, but this thesis does not cover them. The following actions are appointed for improvement:

- **Loading data from file (txt, xml)**
  There are various sources from which data can be loaded. Text-files are used most often, but also have (after loading XML-files, which is used not as often) the worst performance. Loading data from Microsoft Office and loading data using ODBC (Open Database Connectivity) has the best performance, although their performance degrades for larger number of rows. In MagnaView, data loading using the latter methods is done using standard components. From the applications point of view, this is a ‘black box’ and it cannot be improved. The loading of text- and xml-files have a custom implementation. These can be improved.

- **Conversion to internal data structure**
  All loaded data is converted to the internal data structure and saved in memory. The data structure chosen is not optimal and requesting data from this structure can be slow.

- **Creating tree structure**
  The creation of a tree structure is a frequently used action in the application and, although not very slow in itself, this action decreases the performance because of the frequency of its usage.

- **Internal data type conversion steps**
  All data has a type, but this type is not always known. All data is stored as strings and this leads to many type-conversions. Type-conversions are expensive operations and should be avoided as much as possible.

- **Aspect ratio**
  In visualization of data using treemaps the aspect ratio of the rectangles used can be very important. In MagnaView a *squarified treemap*-algorithm [3] is used mostly, because this algorithm yields the best treemaps. This algorithm tries to minimize the aspect ratio (relation between the width and height: the largest from
these values divided by the other value) of the rectangles. The aspect ratio of a
rectangle is defined as:
\[
AR(rect) = \frac{\text{Max}(\text{width}, \text{height})}{\text{Min}(\text{width}, \text{height})}
\]
From this formula it follows that AR ≥ 1. The optimal aspect ratio is 1, i.e., the
width and height of the rectangle are equal, so it is a square.

The algorithm generates rectangles where the aspect ratio is close to 1. It is not
always possible to use just squares, since this violates the space-filling property of
treemaps. This property states that all child nodes together should occupy the
same area as their parent node. For instance, when the parent is a square and has
three child nodes, it is not possible to obtain an aspect ratio of 1 for every child
node.

The calculation of the aspect ratio is an action that is performed very often and its
performance can be improved. This action is not part of the data handling and
therefore will not be covered in this thesis.

- **Rendering cushions**
  One of the last steps in the visualization of the data is the rendering of the
treemaps. MagnaView uses cushioned treemaps [4], because it displays hierarchy
and because this results in attractive images. Cushioned treemaps use shading to
give the rectangles a 3D-look [11]. As a result the rectangles do not have a
uniform colour (luminance), but each pixel of a rectangle can have different
shading. Therefore, calculation of the colour has to be done for each separate
pixel, instead of one calculation for each rectangle. This is very time consuming.
This action is not part of the data handling and therefore will not be covered in
this thesis.

- **Node selection**
  In the resulting treemap the user can interactively explore the treemap and request
information about each rectangle. This information is displayed when a user
moves his mouse over the rectangle. The information is stored in the tree, but
looking up the treemap node that is associated with specific x- and y-coordinates
is expensive. This action is not part of the data handling and therefore will not be
covered in this thesis.

- **Loading Views**
  Next to the actual data there are a number of parameters that determine how the
data is visualized. Among these are parameters for the hierarchical structure and
the colours used. These parameters together are called a view. Users can create
multiple views, each specifying one unique visualization of the same data. All
views are stored together in a file with views, which is an XML-file. This file
needs to be loaded once. The loading process is time consuming. This part is not
part of the data handling and therefore will not be covered in this thesis.
We present performance improvements for the items on this list that are related to the data handling. In particular, we focus on the data loading process and building trees. The performance improvements are presented in the next chapters.
4. Data handling
In this chapter we discuss the data handling part of MagnaView. First, we give an overview of the architecture of the data handling of MagnaView. Next, we discuss the properties of the data and give an overview of data sets.

4.1. Introduction
We now focus on the data handling part of MagnaView. We present the model from the previous chapter (Figure 11) in more detail, but now only the data handling part is shown. This new model can be seen in Figure 13.

![Figure 13 - Model of the data handling part of MagnaView](image)

The first part of the data handling is the data loading. Data can be loaded from various sources, such as text files, XML files, Excel files, Access files and from databases using ODBC. Most commonly used ways to store business data are covered, although currently text files are used most often.

The data that is loaded from the various data sources has a tabular lay-out. To assure the data can be accessed as fast as possible, the data is stored in the application in main memory. The data is stored in an internal data structure. The tabular lay-out is preserved in that internal data structure.

4.2. Formalization
In order to be able to reason about and analyze operations on the data structure, we give a formalization of the data structure. This formalization is based on the formalization as defined in [12]. The columns in the tabular lay-out are called attributes and denoted by $A_i$.
for each column \( i \) (1 \( \leq \) \( i \) \( \leq \) \( N \)). A value for attribute \( A_i \) is denoted by \( v_i \). Each row \( j \) (1 \( \leq \) \( j \) \( \leq \) \( M \)) in the data, denoted by \( R_j \), is a tuple of the values for each attribute.

\[
R_j = (v_{1}, v_{2}, \ldots, v_{N})
\]

The complete representation of the data \( D \) is the set of rows:

\[
D = \{R_1, R_2, \ldots, R_{M-1}, R_M\}
\]

The projection on attributes \( A_{k}, \ldots, A_n \) for row \( j \) is denoted as

\[
\pi_{A_k, \ldots, A_n}(R_j)
\]

and represents the row \( R_j \) restricted to the columns \( A_k, \ldots, A_n \).

The projection on attributes \( A_{k}, \ldots, A_n \) for the complete set of rows (i.e., \( D \)) is denoted as

\[
\pi_{A_k, \ldots, A_n}(D)
\]

<table>
<thead>
<tr>
<th>ID</th>
<th>Car brand</th>
<th>Car type</th>
<th>Engine</th>
<th>Fuel</th>
<th>License plate</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Volkswagen</td>
<td>Golf</td>
<td>1.6</td>
<td>Gas</td>
<td>16-GO-LF</td>
</tr>
<tr>
<td>2</td>
<td>BMW</td>
<td>3</td>
<td>1.8</td>
<td>Gas</td>
<td>18-BM-W3</td>
</tr>
<tr>
<td>3</td>
<td>Volkswagen</td>
<td>Polo</td>
<td>1.0</td>
<td>Gas</td>
<td>10-PO-LO</td>
</tr>
<tr>
<td>4</td>
<td>Mercedes</td>
<td>E</td>
<td>2.0</td>
<td>Diesel</td>
<td>20-ME-RC</td>
</tr>
<tr>
<td>5</td>
<td>Volkswagen</td>
<td>Golf</td>
<td>1.6</td>
<td>Diesel</td>
<td>20-GO-LF</td>
</tr>
<tr>
<td>6</td>
<td>BMW</td>
<td>5</td>
<td>2.0</td>
<td>Gas</td>
<td>20-BM-W3</td>
</tr>
<tr>
<td>7</td>
<td>Volkswagen</td>
<td>Golf</td>
<td>1.8</td>
<td>Gas</td>
<td>18-GO-LF</td>
</tr>
<tr>
<td>8</td>
<td>BMW</td>
<td>1</td>
<td>1.0</td>
<td>Diesel</td>
<td>10-BM-W1</td>
</tr>
<tr>
<td>9</td>
<td>Mercedes</td>
<td>C</td>
<td>1.6</td>
<td>Gas</td>
<td>16-ME-RC</td>
</tr>
<tr>
<td>10</td>
<td>BMW</td>
<td>3</td>
<td>1.6</td>
<td>Gas</td>
<td>16-BM-W3</td>
</tr>
</tbody>
</table>

Figure 14 - Overview of an example data set, together with the related concepts of the data structure.

In Figure 14 an illustration of the tabular layout of a data set and the relating data structure is shown.

### 4.3. Unique Values

The implementation of the data handling part in MagnaView is as close to the model as possible. For the data storage a one-on-one mapping of the tabular input was chosen, because this is the easiest way to implement the model, although this may not be the optimal solution.
In the implementation of the internal data structure the complete data set consists of data records, which are equivalent to the rows in the tabular data. Each data record consists of a number of attribute values, one for each attribute available in the data set. It is important to note that in real world datasets the number of different values for an attribute \( A_j \) is often much smaller than the total number of values (i.e., the number of data records). Below we present statistics to support this. It is possible that the number of different values equals the total amount of values, i.e., all values are unique, so the attribute is an identifier for example, but for most of the attributes the number of unique values is smaller, i.e.,

\[
\# A_j \leq \# D,
\]

where \( \# \) denotes the number of different values.

To optimize the performance and reduce the memory requirements, in the current version per attribute \( A_j \) a list of all different unique values, \( U_j \), is stored. In the internal data structure not the full attribute value, but only the index of the value for \( A_j \) in \( U_j \) is stored.

To illustrate the concept of unique values, we consider a small example. Figure 15 shows the different values for an attribute. In Figure 16 the same values are shown, but now the unique values are stored in a different list and only a reference to the index of the value in the list of unique values is stored.

Using unique values reduces memory requirements, but only if the average size of an attribute value in memory is larger than the size of a reference. If the attribute consists of integer values, the memory size of an attribute value and of a reference are equal and unique values use more memory instead. Most attributes, however, consist of string values, where the memory size of an attribute value is larger, leading to reduced memory requirements.
Unique values can also increase performance. It is possible to define some properties, such as icon or colour for attribute values. The different attribute values that share the same values together form a class of values. Each item in a class should have the same value for a property; e.g. if colours are assigned to car brands, each “Volkswagen” should have the same colour. Unique values define such classes. Instead of assigning a value for a property to each separate attribute value ($#A$ values), it is better to assign a value for a property to each separate class ($#U$ values). This increases the performance, but also joins up to the way users think of data.

For an example of the usage of unique values in visualization tools, we consider a small data set. In Figure 15 the values for an attribute of this data set are shown. Figure 16 presents the unique values for this attribute. For each of the unique values we can assign a colour to be used. This is shown in Figure 17. This is a simplified version of the part of MagnaView that handles unique values and their colours. In this example, we assign separate colours for each unique value. The resulting treemap is presented in Figure 18.

The concept of unique values is not new. Data tools, especially database applications, use database normalization to eliminate redundant data. Although normalization is a process which encloses more steps, it is comparable to using unique values. Both create extra tables or lists to store data that also could have been stored in the original table. A difference is that the extra list of unique values is limited to exactly one attribute, whereas the extra tables created in the normalization process contain multiple attributes.

### 4.4. Data sets

In order to carry out the tests we envisioned, we need data sets to be able to perform performance measurements. In this section we present the data sets used for these measurements and analyze some properties of the data sets. We can distinguish two types of data sets: real world data sets, used by customers of MagnaView, and theoretical data sets created to measure the performance of data sets with special properties.
4.4.1. Theoretical data sets

The first group of data sets consists of theoretical data sets created to make analyses of the performance of the application. We create data sets with special properties to measure the influence of specific properties of the data. The data sets are created with a number of parameters:

- Number of rows
- Number of attributes
- Number of unique values
- Values used

By varying each of these properties, data sets with special properties can be created. It is possible to create data sets with only one unique value, but also with solely unique values. The values used can also be ordered or created as random as possible.

These data sets share that they are not very likely to appear in real world data sets. Nevertheless, they are useful to perform analyses, because the influence of any of the separate properties of a data set can be measured.

4.4.2. Real world data sets

The second group of data sets consists of data sets used by customers of MagnaView. In Table 2 we present an overview of the data sets used for performance analysis. Each data set has an identifier, which is used in this thesis to refer to the data set.

<table>
<thead>
<tr>
<th>Identifier</th>
<th>Rows</th>
<th>Attributes</th>
<th>%Attributes</th>
<th>Min. #U</th>
<th>Max. #U</th>
<th>Avg. #U</th>
<th>%U</th>
</tr>
</thead>
<tbody>
<tr>
<td>Education1</td>
<td>197.516</td>
<td>22</td>
<td>0.011</td>
<td>1</td>
<td>1.427</td>
<td>338</td>
<td>0.17</td>
</tr>
<tr>
<td>Education2</td>
<td>97.738</td>
<td>31</td>
<td>0.032</td>
<td>2</td>
<td>1.562</td>
<td>221</td>
<td>0.23</td>
</tr>
<tr>
<td>Phone</td>
<td>1.173</td>
<td>8</td>
<td>0.068</td>
<td>4</td>
<td>686</td>
<td>266</td>
<td>22.71</td>
</tr>
<tr>
<td>Image</td>
<td>998.783</td>
<td>6</td>
<td>0.001</td>
<td>256</td>
<td>15.158</td>
<td>2.994</td>
<td>0.30</td>
</tr>
<tr>
<td>Contest</td>
<td>2.856.755</td>
<td>19</td>
<td>0.001</td>
<td>1</td>
<td>128.040</td>
<td>7.237</td>
<td>0.25</td>
</tr>
<tr>
<td>Shop1</td>
<td>402.477</td>
<td>26</td>
<td>0.006</td>
<td>1</td>
<td>402.477</td>
<td>28.142</td>
<td>6.99</td>
</tr>
<tr>
<td>Website</td>
<td>3.607</td>
<td>18</td>
<td>0.499</td>
<td>1</td>
<td>3.605</td>
<td>598</td>
<td>16.58</td>
</tr>
<tr>
<td>Travel</td>
<td>50.000</td>
<td>3</td>
<td>0.006</td>
<td>7</td>
<td>50.000</td>
<td>17.141</td>
<td>34.28</td>
</tr>
<tr>
<td>Notary</td>
<td>10.674</td>
<td>25</td>
<td>0.234</td>
<td>1</td>
<td>4.122</td>
<td>928</td>
<td>8.70</td>
</tr>
<tr>
<td>Shop2</td>
<td>48.819</td>
<td>22</td>
<td>0.045</td>
<td>3</td>
<td>38.820</td>
<td>3.662</td>
<td>7.50</td>
</tr>
<tr>
<td>Factory</td>
<td>12.986</td>
<td>22</td>
<td>0.169</td>
<td>2</td>
<td>6.224</td>
<td>602</td>
<td>4.64</td>
</tr>
</tbody>
</table>

Table 2 - Overview of real world data sets

In the overview of the data sets the following properties are listed:

- **Identifier** This is used to refer to the data sets throughout this thesis. The identifier also encloses information about the nature of the data set
- **Rows** The number of rows in the data set
• **Attributes**  The number of attributes (or columns) in the data set
• **%Attributes**  The number of attributes as percentage of the number of rows
• **Min. U**  The smallest number of unique values of all attributes
• **Max. U**  The largest number of unique values of all attributes
• **Avg. U**  The average number of unique values of all attributes
• **%U**  The number of unique values as percentage of the number of rows

From this overview we can see that the sizes of the data sets differ greatly. The smallest data set has 1173 rows and the largest almost 2,9 million. The number of attributes compared to the number of rows is very small, in our overview at most 0,7%. In the overview as well the unique values are shown. The most interesting statistic is the average number of unique values. This is the sum of all unique values per attribute, divided by the total number of attributes. The average number varies from 0,17% to over one third of the attribute values on average. Over all data sets the average number of unique values is close to 10%. The minimum and maximum number of unique values, defined as the smallest and largest number of unique values for any attribute, respectively, also have to be considered, since the unique values determine the performance.

The data set we use and refer to most often is *Education1*. This data set contains grades for students. For each grade there are various attributes available, such as teacher, course and student information. This data set contains on average the smallest percentage of unique values.
5. Data Loading

In this chapter we take a closer look at the data loading step. We analyze the problems in this part and present a new data loading algorithm that solves those problems.

5.1. Introduction

An important step in the visualization of data is the data loading. This part is the largest bottleneck in the visualization process. There are 5 different ways to obtain the data, but text-files are used most often. In testing the speeds of the 5 different connections there are large deviations in performance of each method. Loading data from Microsoft Excel, Microsoft Access and ODBC turns out to be much faster than when it is loaded from XML-files and text-files, although the performance degrades if the number of rows becomes too large.

A possible step in increasing the performance might be using data solely from those three sources. For a user this would require a conversion step for each text-file. This is not optimal for the user, who should not have to deal with such conversions [13]. Although the actual data loading operation might become faster, the time necessary for conversion would reduce the benefit. Moreover, the converted data is not very easy to handle any more. By putting the data in a database the portability of the data would get lost. By converting to Microsoft Access or Excel, one of these applications is needed to modify the data. Text files are very common and can be loaded and modified by many programs.

From the user’s point of view, the data loading step is very important. This action is usually the first action after starting the application. The performance of this action influences the users’ opinion about the application greatly: good performance in this step of the data visualization outweighs a possibly equally good performance in some other step later in the process. To determine the performance in loading text files, we have carried out a series of tests by loading data sets of different sizes. We present the results in graphs like the one in Figure 19.

In this graph the time it takes to load a dataset in seconds is shown as a function of the number of rows. All other characteristics of the data set are kept constant. It can be seen that the function bears resemblance to a quadratic function. For a value of 100,000 rows, the resulting time is 37 seconds. If the number of rows is quadrupled, the resulting time is eleven times as much (411 seconds). This means a user should wait almost seven minutes, before the actual process of visualizing data can start. For an even larger number of rows the data loading time is far too long to work with the application comfortably.
We discuss the data loading performance in more detail. For this purpose we created various datasets. A dataset can be characterized by the following properties:

- The number of columns
- The number of rows
- The data it contains. In particular, the number of unique values for each attribute turns out to be important.

We have generated data sets with different values for each property and we have measured the influence on the performance when varying each of these parameters. The results are presented in the next sections.

5.1.1. Number of columns

In this first experiment we look at the performance of several different numbers of columns. We use a dataset with 1 unique value for each attribute and three different numbers of rows (100,000, 200,000 and 300,000). The results are shown in Figure 20. All results are presented in seconds.
From this graph the relation between the number of columns and the performance of the data loading can be seen; the loading time increases linearly ($R^2 = 0.995$).

5.1.2. Number of rows

In a similar way we can measure the influence of the number of rows on the performance. Again we use datasets that contain one unique value for each attribute and three different numbers of columns (10, 15, 20). The results are shown in Figure 21. All results are presented in seconds.

![Figure 21 - Loading data – effect of the number of rows](image)
This graph shows the relation between the number of rows and the performance of the data loading; the loading time increases linearly ($R^2 = 0.999$).

### 5.1.3. Number of unique values

The last property that has an influence on the data loading is the data itself and in particular the number of unique values for an attribute. We perform the same measurements as above for the same kind of data. For these measurements, we take files with 100,000 rows and 5 columns. The number of unique values differs from 10,000 to 100,000. The results are shown in Figure 22. All results are presented in seconds. In the same graph we also marked the time it would take to load an equally big data set consisting of 100,000 rows and 5 columns without unique values, i.e., where the number of unique values is one. This line represents the (constant) time that is used for the loading process.

![Figure 22 - Loading data - Effect of the number of unique values](image)

For an increasing number of unique values and constant number of rows and columns, we see that the time it takes to load the file bears resemblance to a quadratic function ($R^2 = 0.98$). In this graph we also plotted the amount of time it takes to load the same file without unique values, i.e., where the number of unique values is one. This horizontal line represents the time that can be attributed to the loading process, the actual I/O operations. The area between both lines represents the extra time it takes to handle unique values. For a small number of unique values, the largest part of the time is used for the loading process. When the number of unique values is about 10 percent of the total number of values, the data loading takes up 65 percent of the time. When the number of unique values increases to 40%, handling unique values and loading the data both take
half of the time. Loading data where each and every value is unique takes much more time: over 80% of the time is used to handle the unique values.

As we can see from the measurements, handling unique values is the prime cause of the quadratic loading time; this parameter of the data has the largest influence on the performance. The results of these measurements thus indicate exactly where improvements of the data loading process should be sought: handling unique values.

In real world data sets, the number of unique values for attributes differs greatly. The number of unique values for an attribute depends on the kind of data for that attribute. The number can vary from very few (attributes like gender, grades and continents) up to cases where the number of unique variables can be (almost) the same as the number of rows (unique identifiers, phone numbers and postal codes).

Based on the experiences with real world data sets, as can be seen in Table 2 we can say that in most cases the average number of unique values per attribute is less than ten percent. It is possible, however, that one or more attributes contain many more unique values than this percentage and thus influences the loading process greatly.

We have seen that for small numbers of unique values, the performance of loading data is largely determined by the loading process. Optimizing this part potentially yields the largest improvement in performance for real world data sets. However, we need to keep in mind that the number of unique values can differ greatly and that handling these values is worse than linear in time, so handling unique values should be optimized as well. Ideally, the data itself should have no influence on the data loading and thus the data loading should be linear in time. In the next section, we look at the data loading in more detail and try to find solutions to speed up this loading process. In the end, we perform the measurements as shown above again and compare the results against each other to answer the question whether the desired performance speed-up introduced previously (tenfold performance increase) is achieved.

We started with a version of MagnaView where loading data set Education1 takes almost one minute (version 2.0). At that time there was a more recent version of MagnaView (version 3.0) that was not available for customers; in this version loading the Education1 data set took 8 minutes and 45 seconds. We used version 2.0 as the starting point for our performance improvements.

5.2. Data loading overview

In order to improve the performance, we first have to detect the bottlenecks of the data loading process. We first give an overview of the loading process. Figure 23 shows the contents of the “data loading” part from Figure 13.
This figure shows that the data loading process can be decomposed into four separate steps. First of all the incoming data is parsed. The parsing process contains the scanning of the incoming data and splitting it in the separate values for each attribute. Using these values, lists containing the unique values are built. After the whole file is parsed and all the unique values for every attribute are known, the file is parsed again. The values are now inserted in the data structure using the unique values built previously.

### 5.3. Data loading algorithm

We have shown what the data loading process looks like. Now we show how this process is implemented in the application. For this purpose, we first have a look at the data-loading algorithm. The code to load a data set is shown below in pseudo Delphi code.

```delphi
Function LoadData;
A1 values := GetValuesForNewLine;
A2 for each value v do
A3 AddAttribute(v)
B4 while not End of File do
B5 values := GetValuesForNewLine;
B6 for each value v do
B7 Attributes[column].AddValue(v);
C8 while not End of File do
C9 values := GetValuesForNewLine;
C10 for each value v do
C11 DataRecord.AttributeValues[column] := Attributes[column].IndexOf(v);
```

We give an explanation of the code above. The function `GetValuesForNewLine` is the parsing step in the process. It reads a new line from file and splits this line into a list of
attributes. The first line of a dataset contains the names of the attributes, so the values of the first line are treated separately. These values are used to create attributes. The function `AddAttribute` creates attributes from these values and will make sure that each attribute has a unique name. The rest of the file is read line by line. The values for each line are added to the attribute they belong to using the procedure `AddValue` from the attribute object. This procedure searches if the value has occurred before for that attribute and adds it to the list of unique values if the value is read for the first time.

After this first step, the unique values of each attribute are known. The file is read once more and the actual values for each row are stored in the data structure. For each line in the file (except the first line, which holds the names for the attributes), a `datarecord` is created. This datarecord is an object, which holds a list of attribute indices. This list is populated the second time the file is read. For each attribute, the index of the value that is read from file is searched in the list of unique values and stored in the `datarecord`. After the whole file has been loaded, we have a number of `datarecords` that is equal to the number of rows in the file. For each `datarecord` we have a list of a number of elements that is equal to the number of columns. Each element is an index for the list of the unique values for the corresponding attribute.

### 5.4. Complexity

We now look at the (worst-case) complexity of the data-loading algorithm. All lines of code are denoted by an identifier. We use these to refer to the corresponding operations and determine the complexity for each line of the data-loading algorithm shown in the previous section.

To determine the complexity of lines A1, B5 as well as C9 we give the definition of `GetValuesForNewLine`:

```delphi
Function GetValuesForNewLine
D1 Line = Readln;
D2 for each char in Line do
D3   if char = TextSeparator then
D4     LineValues.Add(String);
D5     String := '';
D6   else
D7     String := String + Char;
D8 LineValues.Add(String);
```

In this function a new line is read and it is split into the different attribute values. The `Readln` function is part of the Delphi standard library and is an $O(c \cdot N)$ operation, where $N$ is the number of attributes and $c$ the average number of characters per attribute. The same applies for the loop that scans the characters (D2). All other operations can be considered constant in time, $O(1)$. The complexity of the `GetValuesForNewLine` function is $O(N)$.

In the data loading algorithm, detecting the attribute names and saving them (A2) is done exactly $N$ times. For each attribute name that is read, the name has to be checked before
saving to prevent duplicates (A3). Worst-case, this is an O(N) operation. Therefore the loop with lines A2 and A3 is O(N^2). The worst-case complexity of all lines of part A of the algorithm thus is O(N^2).

Part B of the algorithm consists of three operations. As we have seen B5 is O(N) and B6 is a loop with M iterations. The function AddValue looks for a value in a sorted list of unique variables and inserts it if not found. Searching is an O(log(U)) operation; inserting is worst case an O(U) operation where U is the number of unique values. The function AddValue thus has O(U) complexity. The total complexity of part B therefore is O(N · M · U).

The last part is almost equal to part B, except for line C11. This line contains a search operation in a sorted list and thus has complexity O(log(U)). The total complexity of part C therefore is O(N · M · log(U)).

The total complexity of the data-loading algorithm follows from the performance of the different parts of the algorithm: O(N^2) + O(N · M · U) + O(N · M · log(U)). In real world data sets, the number of attributes is small compared to the number of rows, as we have seen in Table 2. Where the number of rows can be several hundreds of thousands, the number of attributes is mostly limited to a few dozen. Considering this, it can be said that the number of columns N can be seen as a constant factor in the complexity. The complexity thus becomes O(M · U).

From this complexity, it seems that both the number of rows and the number of unique values are equally important in the performance. However, this does not entirely agree with the performance measurements shown in Figure 22. An explanation for this difference can be given if we look at the exact number of rows or unique values. The number of rows is constant, so operations that depend on this amount always have the same performance. The number of unique values grows during the loading process, so the performance of searching or inserting values in the lists of unique values decreases during the loading process.

The number of unique values differs per dataset. This number always lays between 1 and the total number of rows M. In general, it can be written as a factor of M:

\[ U = \alpha \cdot M \quad \left( \frac{1}{M} \leq \alpha \leq 1 \right) , \]

where \( \alpha \) represents the factor of unique values compared to the total number of rows. The total complexity thus can be written as: O(M · (\alpha · M)). In a worst case scenario, when every value is unique, then this is equal to O(M^2).

### 5.5. Bottlenecks

Using the complexity analysis discussed in the previous paragraph and the benchmarking tools, we have made a list of bottlenecks in the data-loading algorithm. We discuss every
problem and give a suggestion how this could be solved. In the next section we present a solution for these problems. The bottlenecks in the data-loading algorithm are the following:

- **Double loop**
  The data loading algorithm can be divided roughly in two major parts: building the unique values and building the data structure. For each of these two parts the complete data file has to be parsed, as can be seen in Figure 23. These two parsing operations are identical, and thus one of them is overhead; it is not necessary to parse the data twice. Furthermore, building the unique values and building the data structure are two separate operations, but it should be possible to combine those steps and build both of them at the same time. If those steps are combined, then obviously only one parsing step is required. The improvement in the performance is a constant factor, but it can be significant. Since the two major parts of the algorithm are combined into one step, the gain in performance can be up to 50% theoretically.

- **Slow parsing**
  Although the parsing procedure has complexity $O(M)$, the performance can be improved by a constant factor. The current parsing procedure is split up in different parts. First, a complete line is read from the file and stored locally. Next, the characters of this line are scanned and all separate attribute values are stored in a list. The values of this list are used as the actual attribute values and stored in the data structure.

  These steps can be done in parallel. During the loading process, all characters can be scanned and the separate values could be stored in the data structure at once. Executing these steps at the same time reduces unnecessary copying of these string values. String copying is an expensive operation and should be avoided as much as possible. The improvement in the performance is a constant factor, this function has an $O(M)$ complexity.

- **Slow disk read operations**
  When investigating some properties of the performance of the data loading part, it turned out that the number of file input/output operations was abnormally high in relation to the file size of the data set. These operations were caused by the disk read operations in the parsing step. In the parsing step, each line is read separately from file (using unbuffered I/O). This implies that the number of disk read operations in one parsing step is equal to the number of rows in the data set. Reading from hard disk is a slow operation; reading from the internal memory is much faster. Reducing the number of I/O-operations and thus the number of hard disk accesses increases the performance. Instead of reading only one line at a time, larger blocks of data should be read from the hard disk and temporarily stored in memory; all operations that normally read from file would have to use the memory instead. This process is known as buffered I/O.
• *Slow operations on data structure*
  There are some operations that are done frequently, but which have not have the most optimal performance. Every value that is read must be added to the unique values. This requires a search operation of $O(\log(U))$ and if not successful, also an insert operation ($O(U)$), making building the unique values worst case a quadratic operation. Adding values to the data structure has $O(U)$ worst case complexity.

  It is possible to create better performing insert and search operations, but this requires a new data structure, which also can be used to solve the problem of the separate building of the unique values and building the data structure. By improving the data structure, the performance of the data loading can be improved, from quadratic to linear.

Now that we have a list of all bottlenecks in the data loading algorithm, we can improve these issues. Some of them are related to isolated parts of the algorithm, but others require a change in the overall structure of the algorithm and the data structure used. In the next paragraph, we present the new data-loading algorithm that improves the performance and solves the bottlenecks.
6. New data loading algorithm

6.1. Requirements
To solve the problems stated in the previous chapter, we create a new data loading algorithm. There are a few requirements formulated before designing this algorithm:

1. the unique values per attribute, as they are created in the old algorithm, are set as post-condition of the new algorithm. Per attribute, a list of unique values should be available when the algorithm is finished. Unique values are essential for creating a hierarchical structure and building trees.
2. in the new data structure it should be possible to address each item directly. No further search operation should be used, to offer the best performance.

6.2. Overview
We start with a graphical overview of the new data loading process, shown in Figure 24.

As can be seen the data loading starts with parsing the incoming data. The parsing step leads up to two different building processes: building the unique values and building the data structure. The unique values that are built are used in the data structure building step.
6.3. Algorithm

Now we present the algorithm in pseudocode. Note that this algorithm is an abstract version. For the sake of simplicity we omitted some details and error handling, which are present in the implemented algorithm.

```plaintext
Function LoadData
  while not End Of File do
    FillBuffer(Buffersize);
    For each character c in buffer do
      if (c = End Of Line or c = AttributeSeperator)
        attributevalue = getAttributeValueRead();
        Attributes[aNr].AddValueForRecord(attributevalue, rNr);
        aNr := aNr + 1;
      if (c = End Of Line)
        aNr := 0;
        rNr := rNr + 1;
      if (c <> End Of Line and c <> AttributeSeperator)
        UpdateBufferPositions;
```

Now we give a short overview of the algorithm and we make an analysis of the complexity of this algorithm.

First of all, the algorithm uses a buffered read operation. Instead of accessing the file directly for each operation, a portion of the file is read at once and stored in a buffer that is located in the internal memory. This is done by the function `FillBuffer`. All further operations use this buffer, until all characters in the buffer are handled. Then the buffer is filled again and this whole process is repeated until the complete contents of the file are handled. This loop is linear in the size of the file.

The characters in the buffer are scanned to find the special characters. These are the characters between the different attribute values, in the code called `AttributeSeperator`. These characters determine the boundary between two consecutive attribute values. Possible characters that are used for this purpose are comma’s (comma separated files) and tabs. Another interesting character is the end of line character. This character denotes the border between two consecutive lines in the file and thus the beginning of a new row.

If one of those characters is found in the buffer, we know that all consecutive characters in the buffer since the last occurrence of such a special character together make up the attribute value. This value is returned by the function `getAttributeValueRead`. The current position in the buffer and the position of the last special character are recorded, so this function can immediately give the resulting attribute value. Therefore this is an O(1) function.

Next to the buffer positions, which are updated for special characters in `getAttributeValueRead` and for other characters in `UpdateBufferPositions`, we also need to keep track of the current row and the current attribute. The current attribute number is denoted by `aNr` and is increased if an attribute separator is read. If an end of line character is detected, then this value is reset to its default value. The current row number
is increased if an end of line character is read. These values are important in the new data structure and will be used when the attribute value is added to the data structure.

Now we have covered all functions, except the function that adds the value to the data structure. All functions are constant in time and the function \textit{AddValueForRecord} also has a $O(1)$ complexity. This implies that the data loading algorithm only depends on the size of the file and thus has a linear complexity. We discuss the new data structure later.

6.4. \textbf{Solutions to the bottlenecks}

Previously, we have identified a list of bottlenecks previously. We also presented a new data loading algorithm that should solve those bottlenecks. Now we discuss each problem again and show how the algorithm contributes to the solution.

6.4.1. \textbf{Double loop}

The reason the old algorithm has a double loop (first building unique values and then building the data structure) is contained in the data structure that was used. In the main data structure references to the indices of the unique values are stored. The lists of unique values are sorted. Since the indices of the items possibly change in a sorted list whilst adding items to the list, all unique values in the file should be stored in the list of unique values before the actual data structure can be built.

A possible solution to this problem is enclosed in the explanation. If the unique values are not sorted, then the unique values and the data structure could be built at the same time, which would require only one single parsing step. This implies that the parsing time is cut in half. However, keeping the lists of unique values not sorted, decreases the performance of handling the unique values. For every new value, we have to check if it occurs in the list with unique values. In a sorted list this search operation is $O(\log(U))$, whereas searching an unsorted list cannot done better then $O(U)$. Especially for larger numbers of $U$ the difference in performance is big. This decrease in performance does not outweigh the speedup gained by cutting the parsing time in half.

We create a new data structure which performance has no direct relationship with the number of unique values. The unique values are built simultaneously with the data structure, which uses the unique values known so far. The list of unique values is not sorted, thus making inserting new values faster. The search operation, which could become a problem with non-sorted lists as we have seen before, also has become faster due to the operations on the data structure, which we discuss later in more detail.

6.4.2. \textbf{Slow parsing}

The parsing step was slow because the parsing, the scanning and the operation on the actual attribute values were three separate steps. In the new algorithm, these steps occur at the same time, thus reducing the number of string copy operations.
The data is loaded in a buffer and each character is examined. There are special characters that separate attribute values and characters that denote the end of a row. All characters between two consecutive special characters make up one attribute value. The indices in the buffer of the last special character and the current position are stored, so the attribute values can be accessed directly and added to the new data structure. This increases the performance of the parsing operation. Another point to notice is that one of both parsing steps in the old algorithm has become redundant, which also increases the performance.

6.4.3. Slow disk read operations

All read operations on the data in the old algorithm were done using a standard function in Delphi. The function `readln` reads a line in the file and returns this value as a string. Using the documentation, it turned out that the implementation of this function uses unbuffered I/O. We replaced the unbuffered I/O implementation by an implementation that places large chunks of data in a buffer. It uses that buffer to load the actual data in the new algorithm.

Using a buffer from which we read the data character by character proved a solution for another problem. As we have seen in the implementation for `GetValuesForNewLine` in the old algorithm, this procedure reads a line from file and scans every character in the line to detect the different attribute values. Each value is stored in a list and the values on this list are added to the data structure. Using a buffer, we immediately have a list of the characters for each line if we also check for end of line characters. This way we do not need a `Readln` operation for each line, but one operation to fill the buffer with a number of lines at once.

The most important parameter in the buffered I/O algorithm is the size of the buffer, which is denoted by `BufferSize` in the new algorithm. Finding a good value for the size of the buffer can help in improving the performance. This value can be chosen to be any value, but the most optimal choice as size of a buffer is a power of two. The buffer size should not be small, because the performance of the loading process then will be equal to unbuffered I/O. The buffer size also should not be too large, since the buffer has to fit into the internal memory. When the memory usage of the application is too high, the hard disk is used as an extension of the internal memory, which makes the buffer useless.

In Figure 25 the influence of buffer size on loading time is shown. In this graph, the loading time for the `Education1` data set is presented, for several different buffer sizes. The buffer sizes are shown on the horizontal axis, the loading times are shown on the vertical axis. We can see that for growing buffer sizes the performance increases, but at some point, large buffer sizes do not have positive influences any more on the performance. The most optimal size of the buffer is 65536 ($2^{16}$), although the differences are small.
6.4.4. Slow operations on data structure

In the data loading process there are two operations that are performed very often. First, searching a value is important. Each new attribute value has to be searched in the list of unique values to check if this value is read for the first time. If so, this value has to be added to the list of unique values, which is the second important operation. Each value has to be stored in the data structure and possibly also to the unique values. In the new data structure, the function that adds a newly read attribute value to the data structure and the unique values is on average an $O(1)$ operation. The new data structure is discussed in detail in the next paragraph.

6.5. **New data structure**

To improve the performance of the data loading a new data structure is necessary, because the operations on the existing structure cannot be improved and lack performance.

6.5.1. **Requirements**

We are looking for a structure that gives us per attribute:

- a list of all unique values for this attribute. This list does not necessarily have to be sorted.
- a list of for each row a reference to the index in the list of unique values.

If the list of unique values is sorted, then the indices possibly keep changing when adding new values. Having a non-sorted list implies that the indices can be used during the loading process and thus the file has to be read only once. Sorting the list is a relatively
cheap operation and can be done after the loading process is finished, or when a sorted list is needed.

We also want to have a data structure which supports:
- fast search operation
- fast insert operation

### 6.5.2. Data types

There are many data structures possible that meet the requirements, but we are interested in a data structure that offers fast searching and insertion. We now discuss some data types and their properties and try to find the best structure to store our data. [14,15]

#### 6.5.2.1. Array

First of all we could use an array. An array has a fixed size. It is possible to create dynamic arrays, but in the implementation this would mean that arrays have to be resized when the number of elements changes. This is a very expensive operation. Arrays allow to access each element quickly (\(O(1)\)) if the index is known, and inserting is also an \(O(1)\) operation if the value is added at the end of the array. Searching in an array however requires worst case scanning of all elements and thus is \(O(N)\). Deleting an element requires that all subsequent elements have to be moved in the worst case and thus also is \(O(N)\). On average, searching and deleting affect \(\frac{1}{2}N\) elements, so the average case complexity also is \(O(N)\).

Keeping the array sorted at all times increases the performance for searching to \(O(\log(N))\), but insertion will be more expensive: \(O(N)\). A point of notice is that an array has a fixed size. It is possible to create dynamic arrays, but in the implementation this would mean that arrays have to be resized when the number of elements changes. This is a very expensive operation.

#### 6.5.2.2. Linked list

Another structure one can use is a linked list. In a linked list there is no upperbound to the number of elements. In a linked list a pointer to the first element is stored. Inserting, searching and deleting require in the worst case checking all \(N\) elements. On average \(\frac{1}{2}N\) elements have to be checked, making these operations worst-case, as well as average case an \(O(N)\) operation, since direct indexing on a value is not possible. Keeping the list sorted does not optimize the performance due to this reason. Every operation starts at the first element of a linked list and each subsequent element has to be traversed.

#### 6.5.2.3. Binary tree

A special version of a linked list is a binary tree. This is a rooted tree structure, where every element has a link to two elements, instead of one. This structure also has a sorted variant, which is called a Binary Search Tree (BST). When a BST is balanced, i.e., when the depth of any two leaf nodes differs by at most one, then inserting, deleting and searching all are \(O(\log(N))\) operations. This yields for the worst, as well as for the average case. There are several types of BST’s, such as red-black trees and AVL trees,
which all have those properties and differ only on details. The implementation of these structures is not trivial however, whereas other structures are mostly part of standard libraries.

### 6.5.2.4. Hash table

A last data structure we consider is the hash table. Hash tables have one operation which is supported very efficiently; searching a value when the key is known can be done in \( O(1) \) on average, although in the (very rare) worst case the performance can be \( O(N) \). The same applies for insertions. Deletions also can be done in \( O(1) \), but using an algorithm such efficient, might require rebuilding the hash table occasionally. The big disadvantage this data structure suffers from is an inefficient memory usage.

### 6.5.2.5. Overview

We now give an overview of the data types discussed above, together with the complexity analysis of the operations. In Table 3 the worst-case, as well as the average case complexity is shown. The first represents the worst case complexity; the last represents the average case complexity. If both are equal, only one is mentioned. [14,15]

<table>
<thead>
<tr>
<th>Data type</th>
<th>Insert</th>
<th>Search</th>
<th>Delete</th>
</tr>
</thead>
<tbody>
<tr>
<td>Array</td>
<td>( O(1) )</td>
<td>( O(N) )</td>
<td>( O(N) )</td>
</tr>
<tr>
<td>Ordered Array</td>
<td>( O(N) )</td>
<td>( O(\log(N)) )</td>
<td>( O(N) )</td>
</tr>
<tr>
<td>Linked List</td>
<td>( O(1) )</td>
<td>( O(N) )</td>
<td>( O(N) )</td>
</tr>
<tr>
<td>Binary Search Tree</td>
<td>( O(\log(N)) )</td>
<td>( O(\log(N)) )</td>
<td>( O(\log(N)) )</td>
</tr>
<tr>
<td>Hash Table</td>
<td>( O(N) / O(1) )</td>
<td>( O(N) / O(1) )</td>
<td>( O(N) / O(1) )</td>
</tr>
</tbody>
</table>

*Table 3 - Overview of worst case and average case complexity of data types*

Based on these results we choose which data type we use to store the data as fast as possible. We will not have to use delete operations, since we only add newly read values. Searching is the most important operation; we have to search every value to detect whether this value already exists. The number of insertions is equal to the number of unique values; this number is at most equal to the number of search operations, but mostly only a percentage of this value.

Based on the results shown in Table 3, we determine which data type suits best. Looking at the worst case complexity, we might opt for balanced binary search trees. If we also consider the average case complexity, the choice for our data structure is easy. Hash tables offer, on average, the best performance, compared to the other data types. Although the performance of inserting and searching values has a theoretical worst case performance of \( O(N) \), this only is hypothetical. The average performance is \( O(1) \), and this can be reached by clever choices for the hash table size and hash function as we discuss in the next section.
6.5.3. Hashing
Before discussing the usage of hash tables in our data structure, we explain hash tables in more detail. A hash table is a data structure that associates keys with values. All values are stored in an array. The keys are mapped to array positions using a hash function. It is possible that multiple keys are mapped to the same array position, this is called a collision. Based on the hash function chosen, the number of collisions differs. With perfect hashing no collisions occur, but it is difficult to create a perfect hash unless some knowledge of the data is available.

In a good hash function the keys are distributed uniformly over the available array positions, in order to keep the number of collisions as small as possible.

6.5.3.1. Load factor
An important value in open addressing hash tables is the load factor of the hash table. The load factor, or \( \alpha \), is defined as:

\[
\alpha = \frac{\text{Number of occupied positions}}{\text{Total size of hash table}}
\]

The load factor is a value that represents the proportion of the slots that are occupied. The smaller the load factor, the better the performance will be in general. A small load factor however implies many unoccupied slots and thus inefficient memory usage. With a high load factor it might be difficult to find an unoccupied position and this can degrade the performance, because there are many collisions that need to be resolved. In general there are two methods to resolve collisions: chaining and open addressing.

6.5.3.2. Chaining
In chaining the collisions are solved by storing multiple values at the location of an array element. The array element points to the first element of a linked list. If the key is mapped to a slot that is already occupied, than the new value is added at the end of the list. Adding an element to a linked list is in general an \( \mathbf{O}(1) \) operation, but in a hash table each element of the list has to be checked to see if the value already exists. When searching for a value, there are two options:

- the key is mapped to an empty array element. In this case the value does not exist;
- the key is mapped to an array element that contains a linked list. In this case the complete list should be traversed, because there are multiple keys that are mapped to the same array element. In the linked list there should be some information stored about the key that is associated with each value, so the correct value can be found. The performance of searching depends on the load factor of the hash table. Searching takes expected time \( \mathbf{O}(1 + \alpha) \). [15]

In chaining the size of the array that contains the values can be kept small. The performance loss for smaller arrays is relatively small, but the memory is used more efficient.
6.5.3.3. Open addressing

Another collision resolution scheme is open addressing. If a key is mapped to a slot that is already occupied, another position is computed. This process of computing new candidate positions (also called the probe sequence) continues until an unoccupied position is found. This requires that the size of the array should be at least equal to the number of values that has to be stored. Theoretically, searching for an unoccupied position could take up \(N\) probes, but this is only possible if the array contains \(N-1\) elements and the hash function is not optimal. A hash function is considered optimal if the keys are distributed uniformly over the available elements.

Under the assumption of uniform hashing, where each key is equally likely to have any of the possible probe sequences, we can give an expected number of probes for a search operation in the hash table based on [15]:

- For an unsuccessful search, the expected number of probes is \(\frac{1}{1 - \alpha}\)
- For a successful search, the expected number of probes is \(\frac{1}{\alpha} \cdot \ln \frac{1}{1 - \alpha}\)

There are several methods to compute probing sequences. We give an overview of the different methods. In these formulas \(h\) is the output of the hash function and all probes are computed modulo the number of available slots in the hash table.

- **Linear probing**
  This function uses the most basic probing sequence. If a slot is occupied, the next in order is probed. This method has the following probe sequence:
  \(h, h+1, h+2, h+3, \ldots\)

  This method has one disadvantage, it suffers from clustering. A cluster in the array with values is a series of values all closely next to each other, while there are many positions available. Consider the following example from [14], which explains clustering:

  \[
  \begin{array}{cccccccccccccccc}
  0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 & 16 & 17 \\
  \hline
  \end{array}
  \]

  In this example there are 18 slots, of which 8 are occupied. Any new key that will be inserted will go into one of the ten available slots, but the chances for each slot are not equally likely. The hashing function will return any of the 18 slots as index, but if this index is already occupied, the next in line will be considered. If \(h(key) \geq 12\) and \(h(key) \leq 16\), then the key will be inserted in slot 16. The key will go in slot 9 if and only if \(h(key) = 9\). The chance for the key to be in slot 16 thus is five times as high as position 8. In this way, the clusters will be likely to grow even longer.
• **Quadratic probing**
  This method uses a different probing method. It tries to solve the problem of clustering. It uses the following probing method:
  \[ h, h + 1^2, h + 2^2, h + 3^2, \ldots \]

• **Random probing**
  This method also tries to avoid the problem of clustering. It generates a list \( r \) with random numbers of length equal to the size of the hash table. The probe sequence becomes:
  \[ h, h + r_0, h + r_1, h + r_2, h + r_3, \ldots \]

• **Double hashing**
  This method also tries to solve the problem of clustering in another way. If a collision is detected, a second hash function is used to compute another probe: \( h, h + h_2(1), h + h_2(2), h + h_2(3), \ldots \)

### 6.5.4. Hashing in the new data structure

We have to determine how the new data structure is set up. First, we have to determine what the keys and the values in our hash table should be. We want to look up if a value already exists; these values thus are the keys of the hash table. As the value we could use a Boolean value, which represents whether or not this value has occurred before. This however will be ambiguous, since multiple keys can be mapped to the same value and collisions will not be detected. Therefore we have to store the full unique value, so we can unambiguously tell whether the new value is already read and added before.

We do not store this value directly in the hash table. During the loading process, if a new unique value is found, we add this value to an auxiliary array. We have seen that this is an \( O(1) \) operation. We store a reference to this value in the hash table. This is illustrated in Figure 26. In this way, we can directly access the value to compare it against a newly read value that might collide with another value when added to the hash table.

Using a supplementary array to store all the unique values has an advantage. If all values are read, a complete list of all unique values is available. Without this array, this would imply that the complete hash table should be inspected, which is an \( O(H) \) operation, where \( H \) denotes the size of the hash table. A list of all unique values is one of the requirements of the new data structure.
Using an supplementary array of course comes at a small price. This method requires more memory than actually needed. Instead of storing the unique values in the hash table, they are stored in the supplementary array and in the hash tables references to this array are stored. The extra overhead is formed by the references in the hash table. In the worst case, when the number of unique values is equal to the total number of values, this would mean that there are $N$ references per attribute. These references can be small; if implemented using an integer, each reference counts 4 bytes. For a file with 100,000 rows and 10 attributes this would mean that less than 4 MB are needed and only if all values are unique. This small amount justifies using an extra array. On top of that, the extra memory is only needed temporary; after the file is read the hash table can be deleted.

As we have seen there are two collision resolution schemes: chaining and open addressing. As we have seen, these methods differ somewhat on the number of probes in searching. If we choose a good hash function, the number of unsuccessful searches will be small. The expected number of probes for a successful search is smaller for open addressing than for chaining, if the load factor is smaller than about 0.7. We use open addressing as our collision resolution scheme. With this choice we also have to choose a method to find a probe sequence.

The method with the least number of probes on average is double hashing, since this method approaches uniform hashing at best. The number of probes does not differ much if the load factor of the hash table is small. We can make sure that the load factor does not exceed a certain value. This value, $\max \alpha$, can be reached by choosing the size of the hash table to be $\frac{1}{\max \alpha}$ times the total number of values (i.e. the total number of rows). The load factor will not exceed $\max \alpha$, but in most cases this factor will be much smaller. The average number of unique values is often only 10% as we have seen before.
With such a small load factor, the costs for applying an extra hash function are larger than the benefits.

A better choice is to get an easier to compute probe sequence. The easiest method is linear probing. This however leads to clustering for high load factors. The number of collisions will get high and the performance decreases. This can be seen in Figure 27. In this graph, the loading times for a data set with 300,000 rows and 15 columns are shown. On the vertical axis, the loading times are shown in milliseconds. On the horizontal axis, the percentage of unique values in the data set is shown.

The graph shows that the loading times are constant for up to 40% unique values. For larger percentages of unique values, the loading times increase rapidly. This is due to clustering and the increasing number of collisions.

We use a different method for our probe sequence instead. We want to have a method as simple as possible; it should not take too much additional time to compute. To prevent clustering, we use an adapted version of linear probing. Instead of probing the next slot $h+1$, we probe slot $h+c$. The probe sequence thus becomes:

$$h, h + c, h + 2c, h + 3c, \ldots$$

Using a value of $c$ larger than one, we try to avoid the clusters that possibly appear when with linear probing. The value $c$ should be reasonably large and should also be relative prime to the size of the hash table, to assure each slot will eventually be probed in a probe sequence.
6.5.4.1. Hash function

The problems of collisions and how to deal with them can also be minimized by our choice for the hash function. With a good hash function, the number of collisions will be small. The question however arises, what a good hash value is. The definition of a good hash function is a hash function that distributes the uniformly over the available slots. Based on the characteristics of the data, one could create good hash functions. If one knows that the data consists of a series of integers from $P$ to $P+Q$, with a hash table of size $Q$, then the hash function $h$ can be defined as:

$$h(value) = value \mod Q$$

This will distribute the values uniformly over the hash table.

Unfortunately, we do not have any knowledge of the data. The values in real world data sets can differ greatly. Some attributes have values that do not have much resemblance with each other (names), other have some characteristics in common (Dutch postal codes). No matter what values we feed the hash function, the resulting indices for our hash table should be as random as possible. We have to find a hash function that provides the least number of collisions on average.

We use a hash function from Daniel J. Bernstein. This hash function is already old, but over time this function has proven to be a very good hash function. We list this algorithm below:

```plaintext
Function DJBHash(key): Integer;
    For each char c in key do
        result := result*33 + ord(c);
```

This hash function is based on multiplication by the prime number 33. It multiplies the result so far by 33 and it adds an integer value for any character.

We do not use strings as the keys. The tuples with the indices in the lists of unique values for the attributes on each level determine the key of the hash function. These tuples are implemented using an array and therefore we made some small adaptations to the original algorithm.

Using this hash function we can create functions to add a value to the hash table and search the index of a value in a hash table. First, we show the Add function. This function returns an integer with the index in the unique values.

```plaintext
Function HashTable.Add(Value): Integer;
    index := h(Value) mod H;
    while HashTable[index] not empty do
        if HashTable[index] = Value then
            result := HashTable[index];
        else
            index := (index + NextProbe) mod H;
    HashTable[index] := UniqueValues.Add(Value);
    result := HashTable[index];
```
In this function we first calculate an index using the hash function $h$. If an entry in the hash table exists, then we have to check whether the reference stored at that position is a reference to the same value we want to add. If so, we have found the index in the supplementary list and we can return this value. If this reference points to another value, we have a collision and we try the next index in the hash table, based on the value $NextProbe$. This is an integer, larger than 1, which is chosen to be relatively prime with $H$. At some point we have found an empty slot. We add the value to the supplementary list of Unique Values and store the reference in the hash table. We also return this reference.

All operations in this algorithm are $O(1)$. The loop in this algorithm is carried out at most $\alpha \cdot H$ times. This value is equal to the number of values stored in the hash table and in a (very rare) worst case scenario this value is equal to $N$.

The function to search an item is similar to the function to add a value.

```pascal
Function HashTable.Search(Value): Integer;
    index := h(Value) mod H;
    result := -1;
    while HashTable[index] not empty do
        if HashTable[index] = Value then
            result := HashTable[index];
        else
            index := (index + NextProbe) mod FHashN;
    return result;
```

The difference with the function to add a value is that if an empty slot is found, this means that the value is not present in the hash table and we return a non-existing reference (-1).
7. Building trees

MagnaView uses a visualization technique called *treemap*. In a treemaps hierarchical, tree-structured data is visualized. In MagnaView the data that is loaded has a tabular layout and has no hierarchical structure at all. The treemap generator of MagnaView creates treemap visualizations from tree-structured data. Therefore between the data loading and the treemap generation an extra step is needed. In this step, which is called the tree generator, the data is converted into a tree structure.

7.1. Introduction

After the data has been loaded and stored in the internal data structure, the tree generator uses the data to build a tree structure based on the settings defined in the view. The tree structure is characterized by a number of levels, Z, where each level \( L_i \) (1 ≤ i ≤ Z) is associated with an attribute \( B_i = A_{L_i} \). The attribute determines which nodes for that level exist.

The tree-generator creates a tree \( T \) which is a directed, acyclic graph consisting of a set of vertices and edges, i.e

\[
T = (V, E)
\]

The set of nodes \( V \) is defined as

\[
V = \bigcup_{i} V_i \quad (0 \leq i \leq Z+1)
\]

where

\[
V_i = \begin{cases}  \emptyset & i = 0 \\ \pi_{B_1, \ldots, B_i} & 1 \leq i \leq Z \\ D & i = Z + 1 \end{cases}
\]

The set of nodes consists of a node \( n_{root} \) (the empty tuple), a node for each row in the data and for each level \( i \), a node for each unique tuple defined by the projection on attributes \( B_1, \ldots, B_i \).

The set of edges \( E \) contains all edges of the tree. Every node except \( n_{root} \) has exactly one outgoing edge and thus exactly one unique parent node \( p \). Every node on level \( i \), 1 ≤ i ≤ Z has at least one incoming edge and thus one or more child nodes \( c \). The set of edges is defined as

\[
E = \bigcup_{i} E_i \quad (1 \leq i \leq Z),
\]

where
\begin{equation}
E_i = \left\langle c, p \right\rangle \mid c \in V_i \land p \in V_{i-1} \land p \in \Pi_{b_i \ldots b_{i-1}} \left( V_i \right) \right\rangle \quad 1 \leq i \leq Z + 1
\end{equation}

The nodes and the edges of the tree thus are dependent on the levels and the attributes for each level. The order of the levels is important.

### 7.2. Overview of building trees

We now look at an example to explain how a data set is converted to a tree. Consider the following (small) data set of a collection of cars and their properties, shown in Table 4.

<table>
<thead>
<tr>
<th>ID</th>
<th>Car brand</th>
<th>Car type</th>
<th>Engine</th>
<th>Fuel</th>
<th>License plate</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Volkswagen</td>
<td>Golf</td>
<td>1.6</td>
<td>Gas</td>
<td>16-GO-LF</td>
</tr>
<tr>
<td>2</td>
<td>BMW</td>
<td>3</td>
<td>1.8</td>
<td>Gas</td>
<td>18-BM-W3</td>
</tr>
<tr>
<td>3</td>
<td>Volkswagen</td>
<td>Polo</td>
<td>1.0</td>
<td>Gas</td>
<td>10-PO-LO</td>
</tr>
<tr>
<td>4</td>
<td>Mercedes</td>
<td>E</td>
<td>2.0</td>
<td>Diesel</td>
<td>20-ME-RC</td>
</tr>
<tr>
<td>5</td>
<td>Volkswagen</td>
<td>Golf</td>
<td>1.6</td>
<td>Diesel</td>
<td>20-GO-LF</td>
</tr>
<tr>
<td>6</td>
<td>BMW</td>
<td>5</td>
<td>2.0</td>
<td>Gas</td>
<td>20-BM-W3</td>
</tr>
<tr>
<td>7</td>
<td>Volkswagen</td>
<td>Golf</td>
<td>1.8</td>
<td>Gas</td>
<td>18-GO-LF</td>
</tr>
<tr>
<td>8</td>
<td>BMW</td>
<td>1</td>
<td>1.0</td>
<td>Diesel</td>
<td>10-BM-W1</td>
</tr>
<tr>
<td>9</td>
<td>Mercedes</td>
<td>C</td>
<td>1.6</td>
<td>Gas</td>
<td>16-ME-RC</td>
</tr>
<tr>
<td>10</td>
<td>BMW</td>
<td>3</td>
<td>1.6</td>
<td>Gas</td>
<td>16-BM-W3</td>
</tr>
</tbody>
</table>

Table 4 - Sample data set with information about cars

This data set consists of the following attributes: car brand, type, engine, fuel type and license plate number. The column ID is not part of the data, but is used to identify the separate rows in the data set. When this data set is converted into a tree, this tree consists of a root node \( V_0 \), a number of levels with nodes \( V_1, \ldots, V_2 \) and the data records \( V_{Z+1} \).

We now look at a number of trees that can be created from this data set. First, we look at the simplest tree that can be created; this is a tree without any levels. This results in the following tree:

![Figure 28 - Tree structure without levels](image-url)
This tree only consists of the root node and the data records. If we introduce a level in the tree, the resulting tree will be different. The following example (Figure 29) shows a tree with one level. The attribute that is associated with that level is the car brand.

![Figure 29 - Tree structure with one level: car brand](image)

Note that the attribute we specify for a level determines what the resulting tree will look like. This can be seen if we create a tree with one level, which is associated with another attribute. This is shown in the tree of Figure 30, which consists of one level, which is associated with the attribute car type.

![Figure 30 - Tree structure with one level: car type](image)

The resulting tree is different from the tree in the previous example (Figure 29). The total number of nodes and edges is larger.

The following example (Figure 31) shows a tree with two levels. The first level is associated with the car brand; the second level is associated with the car type.

![Figure 31 - Tree structure with two levels: car brand and car type](image)

Note that the order of the levels is important. Reordering the levels will result in completely different trees, as is shown in the following example (Figure 32):
7.3. Implementation overview

We now look at the implementation of the trees and the tree building algorithm in MagnaView. All nodes are objects of the type TreemapNode. A treemap node consists of a value, a parent, a list of all of its children and some other properties that are required when the tree is converted to a treemap in a later stadium.

A treemap node $t$ has a function AddDatarecord, which adds a data record to the tree, in such a way that $t$ is the rootnode of the tree. This function is called recursively in order to create hierarchical relations. This function adds nodes to the tree in a depth first way. If all data records are added to the same treemap node, then this node is the root of the resulting tree. This is also how the tree building algorithm works: each data record is added to the root node.

7.3.1. Old tree building algorithm

We now give the treebuilding algorithm in pseudocode.

```
Procedure BuildTree;
  Rootnode  := new TreeMapNode;
  Rootnode.Value := nil;
  For each datarecord D do
    RootNode.AddDatarecord(D);
```

The function AddDatarecord is shown below in pseudocode:

```
Function TreeMapNode.AddDatarecord(D);
  if CurrentLevel = MaxLevel then
    child := new TreeMapNode(D);
    child.Value := nil;
  for each child of TreeMapNode do
    if child.Value = ValueForLevelOf(D) then
      child.AddDatarecord(D);
    child := new TreeMapNode(D);
    child.Value := ValueForLevelOf(D);
    child.AddDatarecord(D);
```

The algorithm first checks the current level of the tree. The number of levels for the tree is limited and known in advance. If we have reached the lowest level, then the data record
is added as a new child, which has no value at all. The lowest level of the tree thus will consist of nodes without a value, because this level is not associated with any attribute at all. The number of nodes on the lowest level is equal to the number of data records.

If the lowest level of the tree has not been reached, then there is an attribute associated with the level. The value of the data record for this attribute determines where it is stored in the tree; each unique value is one child node. The algorithm has to search whether this value already exists in the list of children. This is an $O(1)$ operation. If the child node that represents the value has been found, then the data record is added to this node, otherwise a new child node is created and the data record is added to this node recursively.

### 7.3.2. Complexity analysis

The complexity of the AddDatarecord algorithm is $O(U)$. This algorithm is recursive however; in each step a deeper level of the tree is handled. The number of recursive calls is equal to the number of levels $Z$ and this value is constant. It might seem that the complexity of AddDatarecord is $O(U^Z)$, but this does not hold. To show this, consider a tree, consisting of two levels. Level one is associated with an attribute where every value is unique, so the number of unique values of $B_1$ is $M$. Attribute $B_2$ is any other attribute.

The root node is connected to $M$ nodes on level one, since attribute $B_1$ has $M$ unique values. Each of these nodes on the first level is connected to exactly one node on level two, since there is only one row in the data where some value of $B_2$ is in the same row as the value of a certain node of level node. In general, if there is some level $B_i$, which is associated with an attribute with $M$ unique levels, then on level $i$ there will be $M$ different nodes. Since the leafs of the tree are made from the data records, we know that there are $M$ leafs and they all have the same depth. This implies that nodes from level $i$ up to level $Z$, have outdegree one. The outdegree is the number of edges directed out of a node.

In terms of the implementation, this means that every node on level $i$ and beyond has exactly one child node, making the search operation $O(1)$ for these levels. In our example we have used an attribute with $M$ unique values, but the algorithm is $O(M)$ for any attribute and for any number of levels. The tree starts with one node at the root, and it ends with $M$ nodes as leafs. The sum of the outdegrees of the nodes inspected at each level, and thus the sum of all lists with children nodes, therefore is $M$, which leads to the $O(M)$ complexity of AddDatarecord.

Each separate data record is added to the root node in order to create the tree. The AddDatarecord function therefore is called $M$ times, which implies that the complete tree building algorithm has $O(M^2)$ complexity.
7.4. Solution

In this paragraph we discuss the solution for the slow performance of the tree building algorithm.

7.4.1. Requirements

The old tree building algorithm has a $O(M^2)$ complexity. The number of rows is an important factor; for data sets with twice as many rows, the performance of the tree building algorithm decreases four times. Since we are improving the performance of MagnaView for large data sets, we require only linear dependence of the size (i.e., the number of rows) of the data set.

Next to this first requirement, we also have a constraint that follows from the implementation of MagnaView. All nodes should be objects from the TreemapNode type, since this will ensure a good integration in the existing MagnaView application.

7.4.2. Overview of solution

One of the requirements for the tree building algorithm is a linear dependence on the number of rows of the data set. This means that each data record that is added to the tree should be added in constant time. Obviously, the resulting tree should not differ in any way. We have to determine at each level for a given list of nodes, which of these nodes holds the correct value for the associated attribute, if there is such a node. This step should be done in $O(1)$ time. We can see a clear parallel with the data reading algorithm.

There we have a value and a list of unique values and have to determine if this value occurs in the list and in which position. The solution we gave there to make this a $O(1)$ step, was to use hashing. We use this method in our solution for the tree building algorithm as well.

For any tree that can be constructed from any data set, the following properties hold:

- The tree has a root node $n_{root}$, which has no parent nodes
- The total depth of the tree is determined by the number of levels. For any number of levels $Z$ ($0 \leq Z$), the depth is $Z+2$
- The number of leaves in the tree is equal to the number of rows in the data set. All leaves have the same depth, $Z+2$
- The path for each node and each data record to the root node is unique.

We use these properties to create a new method to build trees. Each node has a unique path, consisting of a sequence of nodes, to the root node. Since each node represents one value of an attribute, the path can be seen as a tuple of these values. This can be seen as a generalized version of a trie, or prefix tree [16]. In a trie, all the descendants of any node have a common prefix of the string associated with that node. In our structure we store paths from the root node to the current node. Our tree structure has some differences compared to tries. First of all, we do not use strings, but integer values stored in an array.
Furthermore, for each node we store the full path from the root node to that node, because we want every node on the same level to be unique.

For each level $i$, $1 \leq i \leq Z$, the tuple with the attribute values, representing the path from the root node to that level, for any row number $j$ in the data set, is defined as

$$\pi_{B_k \ldots B_n}(R_j)$$

Determining if a node of a value for an attribute already exists on a certain level can be seen as searching for a tuple that corresponds with that node. We use hashing to create a fast search operation. Per level, we store all nodes that are created as the values in the hash table. The tuple will be used as the key. We do not use the full attribute values as elements of the tuple, but we use the indices of those values in the corresponding lists with unique values. Again, the unique values are essential.

As opposed to the usual method of building trees, we do not start at the root of the tree, but we start with the data records. We know for each level how the tuple with the attribute values is defined. We use this tuple to search the corresponding node in a hash table that contains all nodes on that level. If the node does not exist, we create it. We connect this node to the node on the previous, lower level and we work our way up the tree, until we have made a path from the data record to the root node. All operations are $O(1)$, so adding one data record to the tree is an $O(1)$ operation. The tree building algorithm only depends on the number of data records, which implies that the complexity of this algorithm is $O(M)$.

It is possible to create a tree building algorithm with the same time complexity, which builds a tree starting from the root node with a depth first strategy, using the same hash tables. This ‘top-down approach’ is similar to the old tree building algorithm and the approach of most tree building algorithms in general. We use a “bottom-up approach” for our algorithm instead. Although this leads to an algorithm, which differs mainly on the sequence of the separate steps, it has some performance improvements over a top-down approach.

The basic time complexity does not change, but the bottom-up approach makes adding a new data record to the tree possibly faster. With a top-down approach adding a data record is constant in time, there are always as many steps as the number of levels. Since each path from the root node to a data record is unique, we need to traverse every level up to the new data record. With a bottom-up approach we possibly have created a unique path from the root node to the data record earlier, because we can use the property that two data records can share a part of the path from the root node.

To explain the early abortion in the bottom-up tree building process, consider the following example:
We have constructed a tree, consisting of two levels, both containing two unique values: ‘a’ and ‘b’. The data set has eight rows, of which seven already are added to the tree. We now are about to add the last row. Suppose the values for both attributes of this row are ‘b’. In a top down approach we start at \( n_{\text{root}} \) and search the child node that is associated with \( b \). From this node, we search for the child node associated with \( b,b \). Once we have found this node, we can add the data record as child of this node. In our bottom-up approach, we start with the data record and search for the node associated with \( b,b \) at level 2. If we have found this node, we can add the data record as child of this node and we are done. We have constructed a path from the root node to our data record, because there is a data record (data record 7), that shares a large part of the path with data record 8.

This approach yields a better performance in building the tree. In the most optimal tree, where every level is associated with an attribute consisting of one unique value, adding a new data record can be done in one step, instead of \( Z \) steps with a top-down approach. The best case performance of the bottom-up approach thus is \( Z \) times as fast as the top-down approach, whereas worst case, both are equally fast.

We give an example to explain the tree building algorithm. We use the data set with cars and their properties, presented in Table 5.
<table>
<thead>
<tr>
<th>ID</th>
<th>Car brand</th>
<th>Car type</th>
<th>Engine</th>
<th>Fuel</th>
<th>License plate</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Volkswagen</td>
<td>Golf</td>
<td>1.6</td>
<td>Gas</td>
<td>16-GO-LF</td>
</tr>
<tr>
<td>2</td>
<td>BMW</td>
<td>3</td>
<td>1.8</td>
<td>Gas</td>
<td>18-BM-W3</td>
</tr>
<tr>
<td>3</td>
<td>Volkswagen</td>
<td>Polo</td>
<td>1.0</td>
<td>Gas</td>
<td>10-PO-LO</td>
</tr>
<tr>
<td>4</td>
<td>Mercedes</td>
<td>E</td>
<td>2.0</td>
<td>Diesel</td>
<td>20-ME-RC</td>
</tr>
<tr>
<td>5</td>
<td>Volkswagen</td>
<td>Golf</td>
<td>1.6</td>
<td>Diesel</td>
<td>20-GO-LF</td>
</tr>
<tr>
<td>6</td>
<td>BMW</td>
<td>5</td>
<td>2.0</td>
<td>Gas</td>
<td>20-BM-W3</td>
</tr>
<tr>
<td>7</td>
<td>Volkswagen</td>
<td>Golf</td>
<td>1.8</td>
<td>Gas</td>
<td>18-GO-LF</td>
</tr>
<tr>
<td>8</td>
<td>BMW</td>
<td>1</td>
<td>1.0</td>
<td>Diesel</td>
<td>10-BM-W1</td>
</tr>
<tr>
<td>9</td>
<td>Mercedes</td>
<td>C</td>
<td>1.6</td>
<td>Gas</td>
<td>16-ME-RC</td>
</tr>
<tr>
<td>10</td>
<td>BMW</td>
<td>3</td>
<td>1.6</td>
<td>Gas</td>
<td>16-BM-W3</td>
</tr>
</tbody>
</table>

Table 5 - Example data set of cars and some properties

We are building a tree consisting of two levels: Fuel and Engine. The lists with unique values for these attributes are shown in Table 6.

<table>
<thead>
<tr>
<th>Index</th>
<th>Engine</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.6</td>
</tr>
<tr>
<td>2</td>
<td>1.8</td>
</tr>
<tr>
<td>3</td>
<td>1.0</td>
</tr>
<tr>
<td>4</td>
<td>2.0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Index</th>
<th>Fuel</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Gas</td>
</tr>
<tr>
<td>2</td>
<td>Diesel</td>
</tr>
</tbody>
</table>

Table 6 - Unique values for attributes 'Engine' and 'Fuel'

In the tree building algorithm, we do not use the attribute value, but the index of the value in the list of unique values instead. In the initial state of the algorithm, we have one node, the root node. We start with adding the first data record. This yields the situation as in state A of Figure 34. The first data record has the values “gas” and “1.6” for the attributes on the levels. Both indices in the unique values of these attributes are 1. The path from the root node can be denoted by the tuple \(1,1\). We search in the list of nodes of level 2 the node that is associated with \(1,1\). This node does not exist, so it is created and this node becomes the parent of the data record. This is shown in state B. We repeat the process for level one, searching and creating a node for tuple \(1\), leading to state C of Figure 34. Now this node on level 1 can be added as a child of the root node (D), and we have created a path from the root node to the data record. This process is repeated for each data record.
After all data records are added to the tree, we have built the tree. The resulting tree is shown in Figure 35. In this figure we have replaced the values with the tuple of indices of each node with the attribute value, to give a clear overview of the tree.

**Figure 35 - Tree structure with two levels: fuel and engine.**

### 7.4.3. New tree building algorithm

The new tree building algorithm is shown below in pseudocode.

```pseudocode
Procedure BuildTree;
    Rootnode := new TreemapNode;
    for each datarecord D do
        path := GetPath;
        newnode := new TreeMapNode(D);
        AddValueToTree(newnode, path, NumberOfLevels);

This function is pretty straightforward. The root node is created and for each data record a node is created as well. Next the function AddValueToTree is called, which adds each data record to the tree. The function is shown below in pseudocode.
```
Function AddValueToTree(childnode, path, level)

tmpnode := Hashtable[level].getNode(path);
if (tmpnode exists) then
    child.parent := tmpnode;
    tmpnode.addChildNode(child);
else
    tmpnode := new Treemapnode(D);
    tmpnode.Value := getValueForRecord(D);
    tmpnode.addChildNode(childnode);
    childnode.parent := tmpnode;
    Hashtable[level].Add(path, tmpnode);
    if Level > 1 then
        AddValueToTree(tmpnode, path, level-1)
    else
        tmpnode.parent := Rootnode;
        Rootnode.addChildNode(tmpNode);

The function AddValueToTree is a recursive function. For the current data record, it adds the value of the attribute associated with the level to the tree and it maintains the parent-child relations and all other properties of the tree. This function has the following arguments:

- **Childnode.** This is the node on the level below the current level. The algorithm starts on level Z and the childnode for that level is a node, which represents the data record.
- **Path.** This is the full path from the rootnode to the data record. This is an array with Z elements, where element \(i\) is a reference to the unique values of the attribute associated with level \(i\).
- **Level.** This is the number of the level, \(1 \leq Level \leq Z\), where a higher number denotes a larger depth in the tree. The initial value of this argument is \(Z\) and for each recursive call this number is decreased.

The function looks for the node for the given level in the hash table. This is the node corresponding with the path from the root node to that level. If this node exists for that level, than we can create the proper parent-child relations and we are assured we have created a path from the root node to the data record. If the node does not exist however, we have to create it and recursively search the levels above, until we have found an existing node, or we have reached the level below the root node. In the latter case, we add the node for level 1 to the root node.

7.4.4. Complexity analysis

Almost all operations in the AddValueToTree function have a \(O(1)\) complexity. Most of these concern setting values, setting parent nodes, or adding nodes as child node. The first call (getNode for a Hashtable) is a search operation in a hash table. This is worst case an \(O(H)\) operation, with \(H\) the number of elements in the hash table. On average this function however has a much better complexity as we have seen before: \(O(1)\). Therefore all operations are \(O(1)\). This function is recursive however, in the worst case this function is called \(Z\) times. The function AddValueToTree therefore has time complexity \(O(Z)\).
Now we can consider the complete \textit{BuildTree} procedure. This procedure consists of a call to the function \textit{AddValueToTree} for every data record. The time complexity of this function therefore is $O(M \cdot Z)$. The number of levels is very small compared to the total number of data records. Therefore this can be seen as a constant factor, thus making the total time complexity of the \textit{BuildTree} function $O(M)$. If we compare this to the old tree building algorithm, which was $O(M^2)$, we changed the algorithm from quadratic to linear. This leads to a big performance improvement. Especially for large data sets, with many data records, this is noticeable. We expand on this in chapter 8.
8. Results

8.1. Introduction

In the previous chapters we have examined the bottlenecks in the data handling of MagnaView and discussed how these can be solved. We have presented a new data structure and new algorithms to improve the performance. We also have shown, using complexity analysis, what the theoretical improvements are. In this chapter we present the results of the performance improvements, by making a comparison using actual measurements.

8.2. Data loading

First, we discuss loading the data. Equal to the data analysis of the data loading of the original application, we split this part into the different characteristics of a data set: number of rows, number of columns and number of unique values.

8.2.1. Number of rows

In the original data structure, the number of rows was a linear factor in the performance of data loading. Loading data sets with twice as many rows, would take twice as much time. In the new data structure this has not changed as we have seen. We show the loading times of some data sets with varying number of rows. All data sets contain 15 columns and do not have unique values (i.e., every value is the same). The number of rows varies from 50,000 to 500,000. In Figure 36 the resulting graph for the old data structure is shown, in Figure 37 for the new data structure. The vertical axes in the graphs represent the time in milliseconds.
As we can see from these figures, both are linear. The only major difference is the performance. The loading times are shown in Table 7. In this table, the number of rows, the loading time in the old and new data structure in milliseconds are shown. Also, the improvements in performance, relative as well as absolute are shown.

<table>
<thead>
<tr>
<th>Number of Rows</th>
<th>Old</th>
<th>New</th>
<th>Improvement (%)</th>
<th>Speedup factor</th>
</tr>
</thead>
<tbody>
<tr>
<td>50.000</td>
<td>9.837</td>
<td>420</td>
<td>95.73</td>
<td>23.42</td>
</tr>
<tr>
<td>100.000</td>
<td>19.606</td>
<td>820</td>
<td>95.82</td>
<td>23.91</td>
</tr>
<tr>
<td>200.000</td>
<td>39.719</td>
<td>1.695</td>
<td>95.73</td>
<td>23.43</td>
</tr>
<tr>
<td>300.000</td>
<td>59.286</td>
<td>2.534</td>
<td>95.73</td>
<td>23.40</td>
</tr>
<tr>
<td>500.000</td>
<td>99.302</td>
<td>4.266</td>
<td>95.70</td>
<td>23.28</td>
</tr>
</tbody>
</table>

Table 7 - Results of performance improvement with the influence of the number of rows

From this table we can see that the improvement is a constant factor. The new data structure performs over 23 times as good as the old data structure. There are no unique values in these data sets; this is the result of improvements with a constant factor.

### 8.2.2. Number of columns

The number of columns also is a linear factor in both the old data structure as well as the new. We make a comparison of the influence of the number of columns in the old and the new data structure. The influence of the number of columns for the old data structure is shown in Figure 38, for the new data structure in Figure 39. All data sets contain 100,000 rows and only one unique variable. The number of columns varies from 5 to 25.

![Figure 38 - Influence of the number of columns in old data structure](image1)

![Figure 39 - Influence of the number of columns in new data structure](image2)

The number of columns indeed is a linear factor. Just like with varying the number of columns, the only difference in these graphs is the performance. The results are listed in Table 8 - Results of performance improvement with the influence of the number of columns.

68
We can see that the performance improvement is large, on average the performance is improved 24 times. The improvements are somewhat larger for smaller number of columns. This can be explained by our improvements. In the old situation, no buffer was used when reading from file; every line in the data set was read separately. From our results, it follows that loading a data set with 5 columns takes 7.36 seconds, while loading a data set with twice as many columns does not take twice as much time. This is due to the constant time it takes to read the lines separately. Rest of the time is used to scan and parse the incoming data; this amount depends on the size of the data. Doubling the number of columns will double the amount of data that has to be scanned and parsed, but the time it takes to read each line separately is constant.

### 8.2.3. Number of unique values

From our complexity analyses, we have seen that the influence of the number of rows and columns is small compared to the influence of the number of unique values for the old data structure. We have presented a new data structure, in which the performance is only linearly dependant on the number of unique values, instead of quadratic. This implies that the performance improvement should increase for larger numbers of unique values. In Figure 40 the influence of the number of unique values for the old data structure is shown. The vertical axis in the graph represent the time in milliseconds.

<table>
<thead>
<tr>
<th>Number of Columns</th>
<th>Old</th>
<th>New</th>
<th>Improvement (%)</th>
<th>Speedup factor</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>7.356</td>
<td>266</td>
<td>96.38</td>
<td>27.65</td>
</tr>
<tr>
<td>10</td>
<td>13.623</td>
<td>528</td>
<td>96.12</td>
<td>25.80</td>
</tr>
<tr>
<td>15</td>
<td>19.723</td>
<td>828</td>
<td>95.80</td>
<td>23.82</td>
</tr>
<tr>
<td>20</td>
<td>25.947</td>
<td>1.138</td>
<td>95.61</td>
<td>22.80</td>
</tr>
<tr>
<td>25</td>
<td>32.437</td>
<td>1.422</td>
<td>95.62</td>
<td>22.81</td>
</tr>
</tbody>
</table>

Table 8 - Results of performance improvement with the influence of the number of columns

![Figure 40 - Influence of the number of unique values in old data structure](image)
As can be seen in Figure 40, the influence of the number of unique values is not linear. The graph shows the loading times for data sets consisting of 15 columns, with 100,000, 200,000 and 300,000 rows, for an increasing number of unique values. The loading time for a data set with 300,000 rows, consisting of solely unique values, is almost 24 minutes.

The same data sets are loaded with the new data structure. The results can be seen in Figure 41.

![Graph showing influence of unique values on performance](image)

*Figure 41 - Influence of the number of unique values on performance for new data structure*

From this graph we can see that the influence of the number of unique values is linear. The graphs for the data sets with equal number of rows are almost horizontal. The unique values however, still have small influence. If the percentage of unique values grows to 100%, the average number of probes to find an empty slot in the hash table also increases, because the load factor increases.

The performance of the application has improved greatly with the new data structure. Loading a data set 300,000 rows and an equal amount of unique values takes 24 minutes with the old data structure, whereas this can be done in less then 5 seconds with the new data structure. The results for data sets with 300,000 rows and 15 columns are shown in Table 9.

As we can see in the table, the performance improvements increase for a growing number of unique values. Without unique values, the performance was improved by 23 times using the new data structure. For large data sets, with many unique values, the performance improvements are much higher; in our examples with 300,000 rows even up to 300 times.
Table 9 - Results of performance improvement with the influence of the number of unique values

<table>
<thead>
<tr>
<th>%Unique values</th>
<th>Old</th>
<th>New</th>
<th>Improvement (%)</th>
<th>Speedup factor</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>115.935</td>
<td>3.953</td>
<td>96,59</td>
<td>29,33</td>
</tr>
<tr>
<td>20</td>
<td>160.232</td>
<td>3.773</td>
<td>97,65</td>
<td>42,47</td>
</tr>
<tr>
<td>30</td>
<td>225.272</td>
<td>3.781</td>
<td>98,32</td>
<td>59,58</td>
</tr>
<tr>
<td>40</td>
<td>314.681</td>
<td>3.906</td>
<td>98,76</td>
<td>80,56</td>
</tr>
<tr>
<td>50</td>
<td>426.560</td>
<td>3.922</td>
<td>99,08</td>
<td>108,76</td>
</tr>
<tr>
<td>60</td>
<td>535.159</td>
<td>4.125</td>
<td>99,23</td>
<td>129,74</td>
</tr>
<tr>
<td>70</td>
<td>680.453</td>
<td>4.038</td>
<td>99,41</td>
<td>168,51</td>
</tr>
<tr>
<td>80</td>
<td>881.332</td>
<td>4.148</td>
<td>99,53</td>
<td>212,47</td>
</tr>
<tr>
<td>90</td>
<td>1.128.706</td>
<td>4.297</td>
<td>99,62</td>
<td>262,67</td>
</tr>
<tr>
<td>100</td>
<td>1.418.585</td>
<td>4.765</td>
<td>99,66</td>
<td>297,71</td>
</tr>
</tbody>
</table>

Table 10 - Results of performance improvements for real world data sets

<table>
<thead>
<tr>
<th>Data set</th>
<th>File size (kB)</th>
<th>Old</th>
<th>New</th>
<th>Improvement (%)</th>
<th>Speedup factor</th>
</tr>
</thead>
<tbody>
<tr>
<td>Education1</td>
<td>23.798</td>
<td>58.859</td>
<td>2.601</td>
<td>95,58</td>
<td>22,63</td>
</tr>
<tr>
<td>Education2</td>
<td>13.129</td>
<td>33.100</td>
<td>1.789</td>
<td>94,60</td>
<td>18,50</td>
</tr>
<tr>
<td>Phone</td>
<td>109</td>
<td>280</td>
<td>45</td>
<td>83,93</td>
<td>6,22</td>
</tr>
<tr>
<td>Image</td>
<td>26.279</td>
<td>91.905</td>
<td>2.837</td>
<td>96,91</td>
<td>32,40</td>
</tr>
<tr>
<td>Contest</td>
<td>242.645</td>
<td>712.503</td>
<td>35.353</td>
<td>95,04</td>
<td>20,15</td>
</tr>
<tr>
<td>Shop1</td>
<td>38.557</td>
<td>147.311</td>
<td>4.297</td>
<td>97,08</td>
<td>34,28</td>
</tr>
<tr>
<td>Website</td>
<td>430</td>
<td>1.178</td>
<td>144</td>
<td>87,78</td>
<td>8,18</td>
</tr>
<tr>
<td>Travel</td>
<td>1.008</td>
<td>3.741</td>
<td>100</td>
<td>97,33</td>
<td>37,41</td>
</tr>
<tr>
<td>Notary</td>
<td>2.685</td>
<td>5.735</td>
<td>298</td>
<td>94,80</td>
<td>19,24</td>
</tr>
<tr>
<td>Shop2</td>
<td>7.357</td>
<td>18.659</td>
<td>767</td>
<td>95,89</td>
<td>24,33</td>
</tr>
<tr>
<td>Factory</td>
<td>3.641</td>
<td>7.242</td>
<td>319</td>
<td>95,60</td>
<td>22,70</td>
</tr>
</tbody>
</table>

For very small data sets, the improvements are smaller, because the initialization takes a relatively large part of the total time. In this thesis we focus on handling large data sets, since the performance was not a big issue for very small data sets. Nevertheless, even for the smallest data set the performance is improved over 6 times; loading this data set only takes 45 milliseconds. The performance improvement for data set Education1 is 22,63 times. This is more then the tenfold speedup we have set as goal.

8.2.5. Comparison with other applications

In Chapter 2 of this thesis, we compared the performance of various treemap visualization tools and data tools with data handling capabilities. In Table 11 we have
listed the results of the measurements for loading data set Shop2. In this overview, we also included the results for the new version of MagnaView with the new data structure.

MagnaView is the only application for which we can measure the data loading and visualization performance separately. The results for the other applications contain the loading process and the standard visualization visible after loading. Microsoft Excel does not have a treemap visualization, but in this application the data is visualized in a grid structure. In order to make a fair comparison, we also included the measurement for the tree building and visualization of the standard view (i.e. a view without any levels) of MagnaView for data set Shop2.

The results of the measurements are calculated by hand, except for the results of both versions of MagnaView, which are calculated using MagnaView Benchmark.

<table>
<thead>
<tr>
<th>Application</th>
<th>Loading time (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Treemapper</td>
<td>7</td>
</tr>
<tr>
<td>Microsoft Research Treemapper</td>
<td>51</td>
</tr>
<tr>
<td>Panopticon</td>
<td>13</td>
</tr>
<tr>
<td>Excel</td>
<td>2.5</td>
</tr>
<tr>
<td>MagnaView (Old)</td>
<td>19.4</td>
</tr>
<tr>
<td>MagnaView (New)</td>
<td>1.2</td>
</tr>
</tbody>
</table>

*Table 11 - Comparison of the performance for loading data set Shop2 in different applications*

From these results it is possible to calculate the speedup factor of the performance improvements in MagnaView. Even with the rendering of the treemap, the performance for loading the data and presenting the most basic view for this data set is improved over 16 times. This is more than the tenfold speedup we have set as our goal. Another important note is that the performance improvements make MagnaView the fastest application in our comparison. It is even faster than Microsoft Excel, which is a data tool.

### 8.3. Building trees

In this second part of the chapter we discuss the tree building. Contrary to the previous section, we only use real world data sets in our examples as these provide the most realistic measurements.

To test the performance improvement, we use the *Education1* data set. We have created a number of views, each with different properties. The views and their properties are listed below:

1. The first view is the most basic view; it does not contain any levels. When data is loaded and there is no information available about the tree structure (i.e., view file not loaded yet), this is the default view users will see.
2. In this view we use one level. The attribute associated with this level is *gld*, which is used as an identifier for classes. It contains 514 unique values.
3. A view with three levels. The attributes associated with these levels are, starting from the root, *gld*, *teacher* and *town*. 
4. The same three levels, but now in reversed order. The attribute *town* contains 37 unique values.

5. A view with nine levels. The first three levels are equal to the levels of the previous view. In addition, it contains 6 other attributes, such as *result*, *school* and *course*.

We compare the time it takes to build a tree for each of the views using the old tree building algorithm, with the time used in the new algorithm. The results are shown in Table 12. All measurements are shown in milliseconds.

<table>
<thead>
<tr>
<th>View</th>
<th>Old algorithm</th>
<th>New algorithm</th>
<th>Improvement (%)</th>
<th>Speedup factor</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>129</td>
<td>40</td>
<td>68,99</td>
<td>3,23</td>
</tr>
<tr>
<td>2</td>
<td>797</td>
<td>32</td>
<td>95,98</td>
<td>24,91</td>
</tr>
<tr>
<td>3</td>
<td>980</td>
<td>47</td>
<td>94,39</td>
<td>17,82</td>
</tr>
<tr>
<td>4</td>
<td>379</td>
<td>39</td>
<td>89,71</td>
<td>9,72</td>
</tr>
<tr>
<td>5</td>
<td>942</td>
<td>40</td>
<td>95,75</td>
<td>23,55</td>
</tr>
</tbody>
</table>

*Table 12 - Comparison of old and new tree building algorithms*

The results show that, using the old algorithm, the tree building times fluctuate. The numbers of unique values for each level, as well as the number of levels in the tree have influence on the resulting time. View 2 and 3 both have the same attribute associated on the highest level, but this last view has more levels, resulting in an extra 0.2 seconds (or 25%) in the result.

View 3 and 4 both have the same number of values, although the attributes associated with each level are in reverse order. View 3 has an attribute with much more unique values on the highest level then view 4 (514 against 37). The more unique values, the more time it takes to search. Therefore, building the tree of view 4 takes 60% less time.

With the new algorithm, the time it takes to build the tree is almost constant. The number of unique values does not have influence anymore, as we have seen in the complexity analysis of the algorithm. The number of levels however, possibly still does have an influence. As we have shown in the analysis of the new tree building algorithm, the algorithm has an ‘early abort’ option. The algorithm tries to build a path from the root node to the data records. Starting from the lowest level, on each level a node has to be found to create a path. If at any point a node in the tree already exists, we have found a path from root node to data record and the algorithm can be aborted for that data record. The chance for an early abort option depends on the number of unique values; the more unique values, the more nodes will have to be created and thus, the chance that the node we are looking for already exists decreases.

In the treemaps used with real world data sets, the attributes associated with the various levels most often only have few unique values. At each level the number of unique values determines the number of groups that can be compared against each other in the treemap. If this number is large, comparison is no longer possible for the user. Therefore, in practice only attributes with few unique values are used in higher levels.
The results of the new tree building algorithm are almost constant. The absolute differences between them are very small, at most 15 milliseconds. The deviations in the results can be explained by the small amounts of time which have to be measured; on average the tree building process takes 40 milliseconds. The methods used for the measurements have influence on the timings.

The performance improvement of the new algorithm is significant. For the simplest view, view 1, the algorithm is more than three times faster than the old algorithm. More complex tree structures lead to even bigger improvements. The reason is that the new the new algorithm has a worst case complexity of $O(M)$, compared to $O(M^2)$ for the old algorithm. The performance improvement is most noticeable if an attribute with many unique values is associated with the highest level. As we have seen before this is avoided in practice, because it would result in treemaps which keep the user from comparing these values. In our example, we have used views, which are closer to the reality. As we have seen, the performance improvement is in most cases more than the tenfold speedup we have set as goal.
9. Conclusion

There is a gap between data tools with visualization capabilities and visualization tools with data handling capabilities. The latter lack the performance in data handling of data tools, whereas data tools do not offer the best visualizations. In this thesis, we have discussed the data handling capabilities of a visualization tool.

This application, called MagnaView, is a good example of a visualization tool. It offers good visualizations using generalized treemaps, but the performance is not very good. In this thesis we have tried to improve the performance in data handling, especially for large data sets. The goal for the performance improvements was to get a tenfold speedup.

First, all bottlenecks in the performance of the application were determined. To measure the performance, we created a small application, called MagnaView Benchmark. From the list of bottlenecks, we have selected the ones related to data handling: data loading and tree building. For both we have presented improvements.

Data loading is the first step in the program. This action contributes for a large part to the user’s opinion about the performance of the application. The loading time did not only depend on the size of the data, i.e., the number of rows and columns, but also on the data itself. This factor turned out to have even the largest influence on loading time.

The influence on the loading time was due to the unique values. For an attribute in a data set, a value might occur more than once. The set of distinct values makes up the unique values. These values are important. Each unique value represents a group of rows in the data set that share a property. In a treemap, the different rectangles also represent groups with a common property. There is a direct relationship between the unique values and the treemap.

To optimize the data loading, various improvements were made. Parsing of the data was improved and also the separate actions of scanning and parsing were combined to one single step. The performance of reading data from file was optimized, as well as handling unique values. To improve this, a new data structure was created. All important operations, such as searching and inserting, have $O(1)$ complexity. The new data structure is based on hash tables. Whereas all other improvements had a constant factor in the improvement, the new data structure made the data loading linearly dependent on the number of rows, instead of a quadratic worst-case complexity.

Hash tables were also used to improve the performance of the tree building. To optimize the tree building process, we have created a new algorithm. This algorithm uses a ‘bottom-up’ approach, instead of the ‘top-down’ approach used mostly. This also leads to an improvement in the worst-case complexity, from quadratic to linear.

Most improvements come at a price. In general the improvements lead to higher memory requirements. There is a trade-off to be made between performance and memory usage.
In this thesis, we have not focused on memory requirements, but we have tried to optimize the performance. The memory requirements introduced with the new data structure and tree building algorithm are relatively small though.

The solutions presented solve the data handling problem for MagnaView. All the different programs, which make up the complete MagnaView application, such as MagnaView Viewer, MagnaView Designer and MagnaView Web, take advantage of the improvements. However, the solutions presented are not restricted to the MagnaView application. They can be used in any treemap visualization tool. The data loading procedure and the data structure we have presented, can even be used in every visualization tool.

The goal for our performance improvements was to achieve a tenfold speedup. For the tree structure, we have presented a new algorithm, which improves the performance of the tree building with over 10 times in general. The solutions and new data structure presented for the data loading of MagnaView have led to improvements of over 20 times. The larger the data set, the larger the improvements can be. In some cases the improvements can be up to 300 times, or even more. The data handling performance of MagnaView is now better than the data handling of its competitors. It even approaches the loading times of real data tools.

We have shown that improvements in the performance of data handling for a typical visualization tool can be made. These improvements reduce the gap between visualization tools and data tools considerably. Therefore, the answer to the central question of this thesis can be answered affirmative.

9.1. Future work

In this section we give an overview of some possibilities for further research and improvements on MagnaView.

9.1.1. Non data related bottlenecks

In this thesis we have given an overview of all bottlenecks in MagnaView. We have presented solutions for the bottlenecks related to the data handling of MagnaView, but all other bottlenecks should also be solved, to improve the performance for MagnaView even further. The bottlenecks mostly can be tracked down to a very small part of the program. The most important bottleneck is the rendering of the cushions.

9.1.2. Reducing tree building

The performance improvements for the tree building part are small in absolute numbers. However, they are very important, since trees are built very often. For every action that might be related to the tree structure, the tree is completely rebuilt. This is done too often. Only actions that actually make adaptations to the tree structure should result in rebuilding the tree. Changing the colours of the nodes or changing the visual appearance
of the treemap leave the tree structure intact, although even for these actions the tree is rebuilt completely.

9.1.3. Using the new data structure for maps and categories
In MagnaView it is possible to create maps and expressions. Both are related to one or more attributes and both can be associated to a division level in the tree structure. Maps are used to map attributes to categories. An attribute with grades can be mapped to categories “sufficient” and “insufficient” for example. Expressions do not have resulting categories, but use one or more attributes to create a new set of values. If a data set contains the attributes amount and price per item, then an expression could be created like:

\[ Sales = amount \cdot price\ per\ item \]

Maps and expressions are functions on existing attributes. The result also is an attribute, although they are not treated like so. Maps and expressions should use the new data structure. This will simplify the development of MagnaView as well as the clarity for users.

9.1.4. Handling larger amounts of data
As we have seen the amount of data is growing. In this thesis we have presented solutions to handle larger amounts of data. Data sets with up to millions of rows can be used. The exact numbers depend on the amount of memory available for the application. The amount of data to be visualized is growing faster than the average amount of memory available in computers. Data sets will not fit entirely in memory, so only the most important parts should be stored. Other parts have to be stored in other places, such as a temporary database. Monet [18] offers a way of storing data which bears resemblance to the structure used in MagnaView. This database management system kernel uses solely tables with only one attribute, just like the attributes in MagnaView. If, at some moment in future, the new data structure presented is not sufficient anymore, this system might be a good replacement.
References

[1] MagnaView  
http://www.magnaview.nl


[5] Treemap  
http://www.cs.umd.edu/hcil/treemap/

http://research.microsoft.com/community/treemapper/

[7] Panopticon Explorer .NET  
http://www.panopticon.com/explorer/

[8] Microsoft Excel  
http://www.microsoft.com/Excel/

[9] Sequoiaview  
http://www.win.tue.nl/sequoiaview/

[10] AQTime  
http://www.automatedqa.com/products/aqtime/


