UiT The Arctic University of Norway

Faculty of Science and Technology
Department of Computer Science

Slicer

A practical framework for web-based CMV

Øystein Knutsen

INF-3990: Master’s Thesis in Computer Science - September 2021

“This too shall pass”
– *Unknown*

Abstract

Explorative data visualization is a widespread tool for gaining insights from datasets. Investigating data in linked visualizations lets users explore potential relationships in their data at will. Furthermore, this type of analysis does not require any technical knowledge, widening the userbase from developers to anyone. Implementing explorative data visualizations in web browsers makes data analysis accessible to anyone with a PC. In addition to accessibility, the available types of visualizations and their interactive latency are essential for the utility of data exploration. Available visualizations limit the number of datasets eligible for use in the application, and latency limits how much exploring the users are willing to do.

Existing solutions often do all the computation involved in either the client application or on a backend server. However, using the client limits performance and data size since hardware resources in web browsers are scarce, and sending large datasets over a network is not feasible. Whereas server-based computation often comes with high requirements for server hardware and is limited by network latency and bandwidth on each interaction.

This thesis presents Slicer, a framework for creating explorative data visualizations in web browsers. Applications can be created with minimal developer effort, requiring only a description of the visualizations. Slicer implements bar charts and choropleth maps. The visualizations are linked and can be filtered either by brushing or clicking on single targets. To overcome the hurdles of pure client- and server-reliant solutions, Slicer uses a hybrid approach, where prioritized interactions are handled client-side.

Recognizing that different types of interactions have different latency thresholds, we trade the cost of switching views for low latency on filtering. To achieve real-time filtering performance, we follow the principle that the chosen resolution of the visualizations, not data size, should limit interactive scalability. We describe use of data tiles accommodating more interactions than shown in earlier work, using an approach based on delta differencing, which ensures constant time complexity when filtering. For computing data tiles, we present techniques for efficient computation on consumer hardware.

Our results show that Slicer can offer real-time interactivity on latency-sensitive interactions regardless of data size, averaging above 150Hz on a consumer laptop. For less sensitive interactions, acceptable latency is shown for datasets with tens of millions of records, depending on the resolution of the visualizations.

Contents

List of Figures

List of Tables

List of Code Listings

/1 Introduction

The fields of data collection, storage, analysis, and visualization have developed broadly in the past few decades. While the amount of available data has increased, the cost of processing and storage has decreased. In the span 2000-2014 the cost of computational power decreased by approximately 77% per year[1, 2]. The cost of storage has seen a similar development[3, 4]. This trend has made it feasible to collect and analyze data at large scales. But collecting and analyzing data is not only feasible, it is increasingly profitable. The value of collected data scales with size, as illustrated by the law of large numbers: The more data available, the more generalized conclusions can be drawn. Consequently, we have circular incentives for collecting more data: Storage and processing is becoming cheaper, and the value grows with volume. With this motivation, businesses and academics now collect more data than ever before, and the significance of data is acknowledged to the extent that the value of some of the worlds largest companies is not measured by physical assets but rather the information they own[5].

To maximize the value of gathering data, having the right tools to analyze it is critical: The data itself is only as valuable as the insights it provides. As a result, many new databases and processing frameworks focused on analytical performance have emerged, ranging from columnar databases to OLAP-cubes, GPU-accelerated systems, and distributed solutions. These solutions improve query performance compared to more traditional Online Transactional Processing (OLTP) solutions, allowing faster aggregates on attributes and drill-downs across dimensions. Distributed solutions further offers horizontal scaling when additional capacity or performance is demanded.

In general, interfacing with these tools is done through some application programming interface (API) or query language, commonly Structured Query Language (SQL). The set of people who can explore the datasets at will is therefore limited to those with technical skills. Often, those making decisions based on the data are not the ones responsible for managing the infrastructure. Visual interfaces and reports composed of charts and maps are popular methods to provide insight from datasets to decision-makers. Shifting the analytical responsibility over to decision-makers can be done by providing interactive visual tools to explore datasets. Doing so reduces the time it takes decision-makers to get answers to their questions significantly, allowing them to explore more ideas and potentially gain more insights and discover correlations. Using Coordinated and Multiple Views (CMV) is a way for applications to enable such exploration.

CMV consists of multiple visualizations of the same dataset. The visualizations can be of different types and often visualize different attributes of the underlying data. For example, a histogram showing revenue over time and a bar chart showing counts of sold products by category. A key feature is the *coordinated* part of CMV; when one visualization changes, all other visualizations update to reflect the change. Continuing the previous example, this could be that the category "Shoes" is selected in the bar chart. Then the histogram would update to represent the revenue solely from shoe sales over time.

Businesses are aware of Decision Support Systems (DSS) and CMVs value, and these tools have become a market of their own. In industry, we see companies offering what is coined business intelligence (BI) platforms. These solutions connect to database backends and provide visual interfaces to explore data and make customized reports and dashboards. Microsoft has PowerBI[6], Google has BigQuery BI[7], and Tableau[8] is a popular independent solution. By transparently mapping visual selections to backend queries, these tools give great analytical power to non-technical users. However, the tools are usually used by just a handful of people to make reports and static dashboards they share with others. The ones receiving the static reports get to learn the insights of the ones who made them but cannot discover such insights themselves. These dashboards and reports aim to teach something, but the best way to learn is often to see the patterns and correlations by one's own accord. Making CMV applications more accessible could enable more people to understand more datasets.

Web applications are some of the most accessible types of applications today. The utility of a dataset is a function of how accessible it is and to how many. As virtually all client computers have a web-browser, the web is a well suited platform for data visualization and analysis. JavaScript libraries for creating visualizations in the browser are popular exist in numbers[1], further validating the platform.

CMV has been a field of research for a long time, and some have stated that they are a "solved problem" already in 2007 [9]. However, earlier solutions were not made to cope with the size of today's datasets, nor the expectations for usability and interactivity of modern users. When exploring data in CMV, latency is critical. Users are prone to refrain from exploring slow visualizations and instead make examinations with faster response times[10, 11]. Even differences in latency measured in milliseconds effects how many insights are gained from exploration and the users impression of the application[12, 10].

BI solutions can not be used to make stand-alone applications and are limited in performance by their client-server architecture. JavaScript libraries for visualizations are great for making applications but are limited by having to transfer all the data to the client. Most of them are also static and do not support creating CMV. There also exists a few hybrid solutions with promising performance and scalability, but they are lacking in features, are hard to set up and hard to use.

To provide low latency in web-based CMV, one must overcome network limitations and limited computational power in clients. To overcome these challenges, we propose a bottom-up design optimized for the visualizations and queries the front-end will support. Building on earlier works and original ideas, we aim to support the features we miss in existing solutions while still providing low latency. Slicer provides a framework for making high performance web-based CMV for data analysis. The framework is designed to be practical for developers to set up.

1.1 Problem definition

Earlier works[13, 14, 15] have shown the possibility of visual and real-time cross-filtering of large datasets in the browser. The popularity of crossfilter[15], Javascript charting libraries such as Highcharts[16], and various BI-platforms further proves the demand for tools of this sort. While pleased with features in some solutions and the performance of others, we feel there is a gap. The existing frameworks for creating web-based CMV are limited in numbers, and suffer from at least on of the following weaknesses:

1. See for example: `https://d3js.org/`, `https://vega.github.io/vega/`, or `https://github.com/dc-js/dc.js`

- Too high latency on interactions. Purely client-side driven applications struggle to keep up the performance as dataset sizes increase. And purely server-side driven applications suffer from the interactivity being bounded by round trip time (RTT) and bandwidth on top of the servers ability to handle incoming requests.

- High cold-start times (time from entering application to first drawn frame of all visualzations).

- Missing features: Many datasets today could make use of plotting in maps in addition to charts and having filtering in both categorical, continuous, spatial, and temporal dimensions.

- Too demanding of back-end or client computers. We do not want to rely on distributed or GPU-accelerated back-ends, and if the clients are responsible for doing the heavy lifting it severely limits the size and dimensionality of supported datasets. It is important that the solution is practical to set up and maintain, and should ideally work on commodity hardware.

- Limited in how large datasets can be.

- Hard to set up and use (for developers).

We believe that it is possible to provide a practical framework for developing perfomant applications with many desired features. Such a framework would consist of a Javascript library for creating the front-end clients and a backed service that takes in a dataset and works in coordination with the application built by the front-end library. The work should be shared between the client and the server. This should be done in a way that minimizes the need for client-server communication when handling user interactions. Ideally, most user actions can be handled without relying on the backend service. Such a solution should be able to handle visualization and interactive filtering relational data with both categorical and continuous dimensions as well as visualizing data in maps in a meaningful way.

Performance can be gained by placing computational power requirements on back-end servers, e.g., requiring CUDA compatibility or distributed setups. However, we aim for the usability of crossfilter and believe that ease of setup and use is critical for the framework to be adopted. We propose that it is possible to improve usability and extend the features of existing frameworks for making CMV web-applications while keeping state of the art interactivity. Specifically, the thesis is:

An easy to set up and use framework for creating feature-rich and performant CMV web-applications can be made by using and extending existing web technologies and data handling methods.

1.2 Targeted Applications

In this thesis we aim for a framework to build web-based data analysis applications with state of the art levels of interactivity, while adding features. Without relying on GPU-acceleration, we do not believe it is manageable to achieve this for enormous or unstructured datasets. The targeted applications for the framework therefore have datasets of relational data with records in millions, in contrast to some other works supporting billions[14, 13].

Applications made with Slicer should work in most popular web-browsers (Google Chrome, Firefox, Safari, and Microsoft Edge). The back-end service should be compatible with both Linux and Windows systems from the past decade at least, with no requirement for a GPU.

1.3 Assumptions and Limitations

Striving for maximum performance dictates some assumptions and limitations. When utilizing the newest technologies, backward-compatibility for hardware and software cannot be guaranteed. The thesis's primary focuses also eliminate certain features and considerations from being implemented. Nonetheless, we aim to make it possible to implement features that do not require changes to the underlying data structures in the future. Performance and extendability will be key concerns for the implementation of Slicer, and specifically we make the following limitations on the scope of the thesis:

- We will not consider clients with small screens and mobile devices (e.g., phones) when designing the front-end components.
- We will not consider the security aspects of the framework.
- We will not implement support for dynamically adding data.
- We will only consider relational datasets when implementing the core functionality.

1.4 Methodology

The methodology of the thesis springs from the ACM Task Force on the Core of Computer Science's definitions of computing as a discipline[17] and the framework for information systems (IS) research as presented be Hevner et al. in 2004[18].

In their 1989 report, the *Task Force on the Core of Computer Science* formed by the Association for Computing Machinery (ACM) and the Institute of Electrical and Electronics Engineers (IEEE) introduce three paradigms as the foundation of computing as a discipline:

- **Theory** is the first paradigm and is rooted in mathematics. In this paradigm a mathematician iterates four steps until arriving at a valid theory:

    1. *Define objects of study*
    2. *Hypothesize relationships between the objects as theorems*
    3. *Determine if the theorems can be proved or disproved*
    4. *Interpret the findings*

- **Abstraction** is the second paradigm and is rooted in the scientific method. In this paradigm a scientist also iterates through four steps, but here to investigate a phenomenon:

    1. *Construct hypothesis*
    2. *Make a model and a prediction*
    3. *Execute experiments and gather data*
    4. *Interpret the findings*

- **Design** is the third and final paradigm. Design is rooted in engineering and is concerned with solving specific problems. The paradigm is also iterative and consists of four steps:

    1. *State requirements*
    2. *State specifications*

3. *Design and implement a system*

4. *Test the system*

In the 2004 paper *Design Science in Information systems research*[18], Hevner et al. complete framework for conducting IS research. The paper identifies two main paradigms characterizing IS research: behavioral science and design science. The behavioral science paradigm "*seeks to develop and verify theories that explain or predict human or organizational behavior*", while the design paradigm concentrates on "*creating new and innovative artifacts*". The presented framework encompasses both paradigms and ties them together. With nested iterative research cycles the intended environment is continually taken into consideration while existing knowledge is visited and applied when relevant possible. This interplay is illustrated in Figure 1.1.

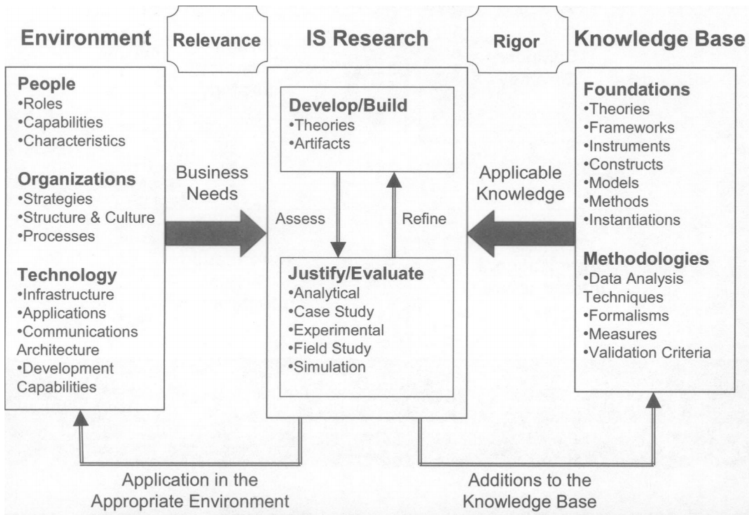


Figure 1.1: The framework for Information Systems Research presented by Hevner et al. in their 2004 paper.

In this thesis both the *theory* and *design* paradigms from the task force's definition have been applied in combination with the more complete cyclical approach of the IS research framework. Where it has been possible *theory* iterations have been used to avoid implementing data structures that are not mathematically sound. With mathematically sound findings added to the knowledge base, *design* iterations were done to implement concrete components in code. These iterations were conducted, as shown in Figure 1.1, with frequent reviews of the relevant knowledge bases (existing technology and literature). The *Environment cycle* of the framework, seen to the left in Fig. 1.1, has been omitted. Business needs were already established and the requirements were known at the start of the thesis, making the cycle irrelevant for this work.

This thesis presents the product of numerous iterations of different research cycles, where Slicer is the final artifact. We present existing knowledge found to be relevant, our own design and implementation, evaluations of said implementation, and conclude with our findings and potential courses for further research.

1.5 Context

This thesis is motivated by challenges I have met while working at SINTEF with one of my advisers, Peter Haro (SINTEF Nord).

At SINTEF, we have many projects where data is gathered and processed, and almost always, the customer and domain experts want to examine the data. As a result, we have tried most existing solutions for making web-based CMV and gone so far as to extend and modify them to suit our needs. The feedback on projects where we have done so has been very positive, but the experience of making them has been problematic. Struggles with limitations in performance, features, scaling, and usability of existing frameworks are what formed the idea for Slicer and this thesis.

1.6 Contributions

The main contributions of this work are:

- Models for
  - Data tile design - Smaller data tiles, requiring less computation to create than earlier work.
  - Dual protocol communication to reduce latency, while keeping throughput for data transfer.
- Methods for
  - Data tile filtering - Using data-differencing, constant time filtering with data tiles is kept, while adding support for single bin selections as well as range selections.
  - Data tile computation - Using database indexing, auxiliary tables, and cached tiles, data tiles can be computed on consumer hardware

    without GPU-acceleration for large data sets.
  - Visualizations - Showing filterable geographical data based on data tiles.

- Artifacts
  - Slicer Framework

1.7 Thesis Outline

The rest of this thesis is structured as follows:

Chapter 2, Background outlines concepts required for understanding the premise for this thesis as well as the related works. The chapter introduces the types of visualizations and interactions with which this thesis and related work are concerned. We look at the underlying principles, data reduction and aggregate functions, and common optimization techniques.

Chapter 3, Related Work provides a review of "state-of-the-art" technologies for data exploration. We present supporting technology and complete frameworks, showing the techniques used to achieve their performance.

Chapter 4, Design presents the design of Slicer and its architecture.

Chapter 5, Implementation describes the implementation of Slicer, focusing on its unique aspects.

Chapter 6, Evaluation outlines experiments and their results before presenting evaluations of Slicer.

Chapter 7, Discussion discusses the choices of Slicer's design and their impact on performance, features, and practicality.

Chapter 8, Conclusion summarizes the thesis and proposes avenues for future work.

2 Background

This chapter provides an overview of techniques and technologies needed to understand the basis of the thesis and existing solutions. We will explain CMV and common techniques used to ensure performance in data analysis and web applications.

2.1 Data reduction

Achieving low latency when interactively visualizing large amounts of data is often done with one or more strategies to reduce the dataset. Query response time depends on data volume, and if the reduction leads to transferring less data between a server and client, bandwidth imposed latency is also reduced. Data reduction in this context means transforming a dataset in some way that reduces the number of data points stored, in contrast to compression, where the goal is to store the dataset efficiently without changing it.

2.1.1 Sampling

Sampling is one of the simplest forms of data reduction. To reduce a dataset with sampling, a predicate function to sample by is chosen and all records not fulfilling it are disregarded. The function can be as simple as checking if the index modulo 10 is equal to 0, thereby reducing the dataset to 10% of its original size. This is a considerable reduction, but it comes at an accuracy cost. It is simply not possible to answer all queries correctly when the whole dataset is not used.

Not all sampling functions are as simple as modulo, however. BlinkDB [19], pre-computes smarter samples to provide SQL support scaled for big data. Doing so, they out-compete their (cluster computing based) competition by a factor of 10-200x, while maintaining reasonably accurate answers (within errors of 2-10% on a 17TB dataset)[20].

2.1.2 Binning

Binning is a data reduction method where data values are assigned to bins. These bins represent an interval in the range of the underlying data values — that way, the values may be stored as representations of the bins. This limits the bytes necessary to represent a single value to the bytes needed to represent the number of bins. E.g., if a dataset containing 64-bit values gets binned into 255 bins, the data values can be stored as single bytes instead of 8.

Bins do not need to represent regular intervals. They can be irregular and multidimensional as well. Similar to how the range 1..100 can be split into ten even bins at an interval of ten, a bin can represent an irregular 2-dimensional area - for example, a country in a 2d map.

If the intervals for the bins are not known, they can be set with clustering algorithms. Clustering algorithms can group similar items usually into a pre-determined number of bins (classes). Creating bins this way will often lead to a higher granularity of bins in ranges with more data as bins are not "wasted" on empty intervals.

Some domains have existing natural bins. An example is geographical data; geographical data often makes sense to group based on country, city, bus-stop, region etc.

2.2 Aggregate functions

In the domain of databases, aggregate functions are functions that reduce a set of values to a single value [5, p.91]. These sets will often be a column in a database, or a selection of rows in a column. Which aggregate functions are natively supported varies between databases. Nevertheless, in SQL-databases at least these five are commonplace:

- **Count**: Returns the number of items in the input set.
- **Sum**: Returns the sum of all values in the input set.
- **Average**: Returns the average value found in the input set. Usually implemented as $Sum/Count$
- **Min**: Returns the minimum value found in the input set
- **Max**: Returns the maximum value found in the input set

In other domains, such functions are also referred to as *reduce functions* with single value outputs. In functional programming, these functions are called *fold* functions and are functions that take a traversable data structure and a combine function (e.g. add) as input parameters. When called , it traverses the data structure and, using the combine function, builds the return value. The returned value may be a single value or another data structure. In the context of this work, all mentioned reduce functions will reduce to a single value. *Aggregate*- and *reduce*- functions are thus interchangeable terms in the context of this thesis.

2.3 Data visualization

2.3.1 Binned aggregates

Binned aggregates are the values produced by binning data and then computing an aggregate on each bin. This computation forms the basis for a subset of data visualizations. These visualizations are especially useful for large datasets since the space they occupy depends solely on the resolution of the visualization, and not the dataset. Common binned aggregate visualizations, as seen in Figure 2.1, include: *bar charts, pie charts, binned heatmaps, and choropleth maps.*

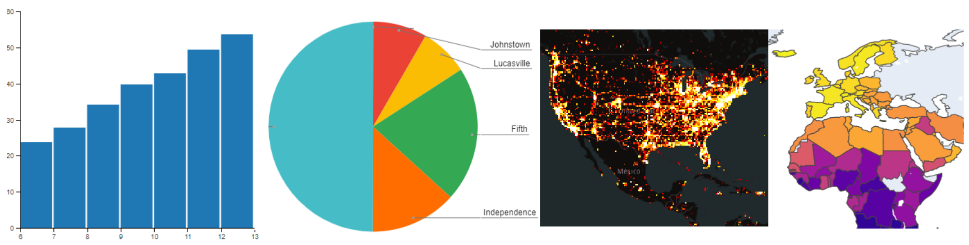


Figure 2.1: Bar chart, pie chart, binned heatmap, and choropleth examples.

2.3.2 Overplotting

Overplotting is the phenomenon that occurs when data points overlap in a visualization - obscuring each other and making it impossible to know how many points are in each location. It is a very common problem when visualizing large datasets, and needs to be taken into consideration when creating views. A screen with FullHD resolution[1] has $1920 \cdot 1080 = 2,073,600$ pixels. This means that if a dataset with more than ~2 million data points is to be visualized on such a screen, overplotting is guaranteed to be a problem even if each data point only occupies a single pixel each.

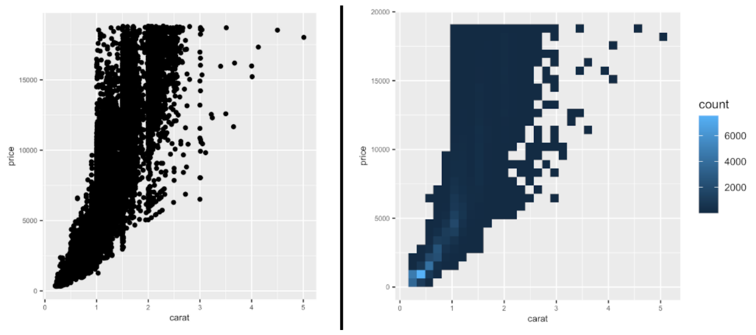


Figure 2.2: Example of a scatterplot suffering from overplotting (left) and the same data in a binned scatterplot to overcome the problem (right). Images are taken from: `https://r-graphics.org/recipe-scatter-overplot`

2.4 Coordinated and Multiple Views

Coordinated and multiple view systems are systems that offer multiple visualizations (views) of the same data. The views are coordinated, automatically reflecting updates to one view in the other views. Exploratory data visualizations made this way allows users to interactively explore their data, potentially discovering valuable relationships and facts that would not be seen with similar non-interactive or non-coordinated visualizations[9].

As outlined by J.C. Roberts in [9], there are vast amounts possible views and ways of coordinating them. Among the nuances, many efforts focus on *scientific visualization*, e.g., volumetric rendering, where one data point usually maps to one pixel on the screen. We, however, are interested in general data visualization and *exploration*. Therefore, we will limit our use of views to charts

1. FullHD is the most common resolution for desktop monitors. See: `https://gs.statcounter.com/screen-resolution-stats/desktop/worldwide` for more statistics on screen resolutions.

(histograms, bar charts, line charts, and scatter plots), tables, and data overlaid on world maps in the form of heatmaps, choropleth, or icons.

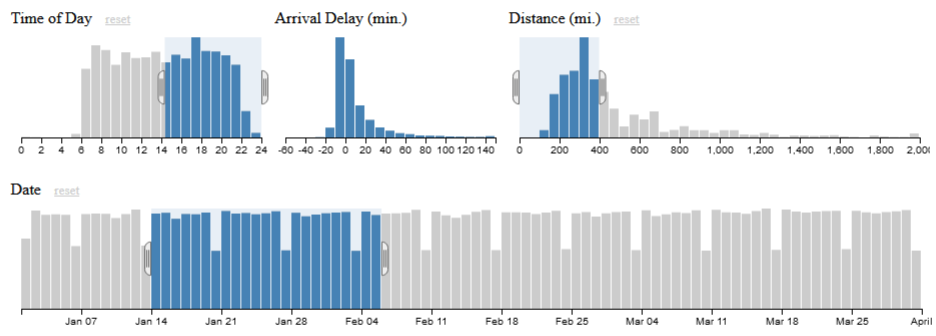


Figure 2.3: Screenshot of example CMV application made with Crossfilter and D3 with bar charts and *linking and brushing*. Can be seen online at: `https://crossfilter.github.io/crossfilter/`

Linking and brushing is a popular method to provide interaction and coordination to visualizations, and it is the primary method used in Slicer and related works. Brushing is the act of selecting a range or an area in a chart by dragging with the pointer. This selection represents a filter in the underlying dataset, and the linking is that the other charts in the application update to reflect this. A brush selection represents *range* filters. They only enable the selection of elements next to each other - e.g. consecutive bars in a bar chart. In this work we are interested in *single-select* filtering also. Single selections enables filtering of multiple elements which are not positioned next to each other. For example if the x-axis in a bar chart represents countries, it makes sense to be able to select two (or more) countries without them having to be placed consecutively in the bar chart.

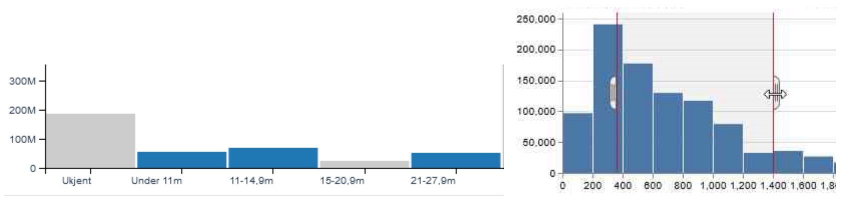


Figure 2.4: Example of single select in bar chart (left) and brush select (right). On the left, the blue bars have been selected by clicking on them, while on the right the selection is the range between the brush edges.

Importance of low latency

Users have shown to prefer interactions with low latency, to the extent of avoiding slow interactions[12, 11]. For CMV applications using linking and brushing, the implication is that all charts should ideally respond to filtering with the same latency. A typical threshold for interaction latency has been 500ms[14], but this level of latency has shown to impact user performance in interactive applications[12, 11]. Another finding by Liu and Heer [11] is that different actions have different sensitivities to latency. An example of a latency sensitive interaction is *brushing*, while *zooming* in a chart is an example of a less sensitive interaction.

For users to get the most value from data exploration similar actions should have similar latency, and the importance of how low the latency is differs depending on action type.

2.5 Optimizations

To achieve fast cold start times and low latency on interaction, many optimization techniques are used in web applications. Here we mention some of the most widespread relevant to this thesis: Caching and progressive loading.

2.5.1 Caching

Caching is a general optimization technique in computer science. Caching is to anticipate what data will be requested in the future and storing it in a more accessible way than the rest of its kind. This anticipation function will vary widely based on the application. Common functions are to cache recently or commonly used values or values stored in proximity to recently accessed values. The advantage of caching comes when it is faster than recomputing values or reading them from a slower data store. Caching is used on most levels of computing, but an example for a web application could be a web-server for a news website storing todays news in-memory, and older news on disk.

2.5.2 Progressive loading

In web applications progressive loading is a technique that improves loading times by deferring non-essential resources. The time from a web app is requested until the first meaningful frame is drawn plays a big part in how responsive it feels. An example for a simple website can be that everything that will not be visible before scrolling down on the page, is not loaded before everything that will be immediately visible has loaded and rendered. Another form of progressive loading is to first load a lesser version of the resource to be able to display *something* immediately, while the actual resource loads in the background. A common use of this form is to load low resolution images and replace them later with the full resolution ones.

In a CMV context, visualizations can be progressively loaded by first showing non-interactive versions and then add interactivity when the necessary resources have loaded.

Progressive loading does not only apply to the initial loading of an app. Any time new resources are requested, the non-essential ones can be down-prioritized or lesser versions can be loaded first.

2.6 Prefix sum

Prefix sum arrays (also called cumulative sum) is a relevant datastructure as it can be used to speed up an operation frequently used when visualizing binned aggregates. Summed area tables[2] are the 2-dimensional generalization of prefix sum arrays. The 1-d and 2-d variants of prefix sums are easiest to visualize and reason about, but prefix sum arrays generalize to n dimensions.

Input Sequence	3	5	3	1	0	2	7	6	...
Prefix Sum	3	8	11	12	12	14	21	27	...

Table 2.1: Example of a 1-d prefix sum array based on a source array.

In a prefix sum array each value in the array represents the sum of all previous values in the sequence so far. I.e. to get the value at index n of the original array, one takes the value at index n and subtract the value at index $n - 1$, unless the index $= 0$, the value is the same in the prefix sum array and the source array.

The operation that prefix sum arrays speed up is to compute the sum of subsequences. When computing the sum of values in the range 2..8 on a regular array, it will require 6 add operations. Using prefix sum, this can be reduced to a single subtract operation, subtracting value 2 from value 8. Thus it changes the complexity of the operation from $O(n)$ to $O(1)$, where n is the length of the sub-sequence. This makes prefix sums particularly useful when plotting binned values as it usually involves summing up the values or counts belonging to each bin.

2. Also called integral images and is used in computer graphics to speed up shading operations and to do object detection.

3 Related Work

This chapter provides an overview of existing works with similar goals to this project. There has been done an overwhelming amount of work done on databases and data visualization. Here we focus specifically on work that enables making CMV data exploration applications with web-browser clients. We do not try to give complete descriptions of all the topics covered, but focus on what makes them related to Slicer and this thesis. For more in-depth descriptions reading the references papers is highly recommended.

3.1 Databases

The increasing interest in data collection and analysis has led to much research on databases. This work has many branches, but we limit our scope to relational databases.

Data visualizations often reflect some aggregate, which relational databases usually support computation of as native features. In this section, we briefly cover developments enabling fast computations of aggregates. The overview provided is limited, as most databases are made to be front-end agnostic, while today's highest performing CMV frameworks rely on purpose-built data structures and encodings residing on the front-end. The relevancy of databases lies in their ability to facilitate the production of those data structures. Additionally, databases might be able to handle queries the custom data structures cannot and thus work in unison with them to achieve more features in the CMV system.

Columnar Databases

Data tables in relational databases are generally stored in one of two types of layouts: *row-oriented* and *column-oriented*[5, p.611]. These layouts are illustrated in Figure 3.1.

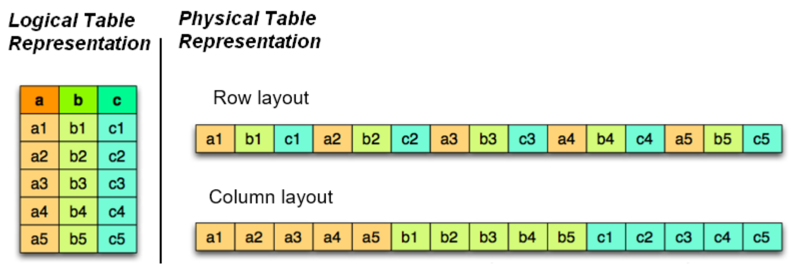

Logical Table Representation
Physical Table Representation
a b c
a1 b1 c1
a2 b2 c2
a3 b3 c3
a4 b4 c4
a5 b5 c5
Row layout
a1 b1 c1 a2 b2 c2 a3 b3 c3 a4 b4 c4 a5 b5 c5
Column layout
a1 a2 a3 a4 a5 b1 b2 b3 b4 b5 c1 c2 c3 c4 c5

Figure 3.1: Illustration of how a table can be stored either *row-oriented* ot *column-oriented*. Source: `https://datacadamia.com/data/type/relation/structure/column_store`

Column orientation is favorable when one or a few attributes from each row is needed in the query. In a column oriented layout, only relevant attributes are read from the database. Whereas row orientation will read entire rows - including irrelevant attributes. This can lead to a significant reduction in disk I/O.

Column orientation can also increase the amount of cache-hits in the CPU[1], further increasing performance. When the CPU requests data from main memory or disk, the CPU will fetch enough bytes to fill a cache line. If the data is fetched from a column oriented database, the extra bytes in the cache line will be the value of the next row in the column. In contrast, if the data is fetched from a row-oriented database, the extra bytes will be the rest of the attributes in the current row. If the query is doing an aggregate on a certain column, these extra bytes will be wasted when fetched from a row oriented database, whereas it will reduce the times the CPU needs to fetch data if it comes from a column-oriented database.

These advantages make column oriented databases useful when many aggregate functions are used - particularly when the aggregate functions use a single

1. A cache-hit happens when data requested is found in cache and does not need to be read from higher up in the memory hierarchy. Modern CPUs have multiple levels of internal caches they try to read from before main memory or disk.

column.

In-memory databases

An in-memory database is a database that is loaded into or created in main-memory, in contrast to being stored on disk. In-memory databases mainly exist to optimize performance. Main-memory has more bandwidth and faster access times than disks[5, p.616]. Additionally, on disks data is stored in blocks further complicating data access. Accessing records in blocks involves getting block identifiers and offsets before checking if the block is in the buffer[2] (and if so, where in the buffer) and finally following the retrieved record pointer to read the data. In main-memory however, accessing a record is done as a regular pointer-traversal, which requires much fewer CPU cycles to complete and is generally a well optimized fast operation on CPUs.

Data Cubes

For a dataset represented as a table of n columns, a datacube for the same dataset is a n-dimensional cube. Although they are called cubes, there can be more or fewer dimensions than 3 and they do not need to be of the same size. Each cell in a data cube represents an aggregate on the intersection of the distinct values of the dimensions at its coordinate. In Figure 3.2 for example each cell represents the number of sales of a category of products, in a specific region, in a specific year. Dimensions and aggregates are computed when the cube is created, making if fast to get out specific aggregates later. One operation that data cubes are effective for is *slicing*. Slicing is the operation of looking at aggregates for a fixed value of a dimension. For example in Figure 3.2, fixing the year dimension on 2004 would produce a 2d *cube slice* containing only aggregates for 2004[3]. When multiple dimensions are fixed, or dimensions are fixed on several bins (e.g. by selecting 2004 and 2005), the operation is called *dicing*[5, p.530].

Data cubes do have some drawbacks, mainly in flexibility and size complexity. In terms of flexibility, data cubes need to be recomputed if changes are made to the dimensions of the underlying data. If different types of aggregates are wanted on different dimensions, data cubes also lose their merit. The space complexity of data cube can be very large compared to the underlying data

2. Databases make use of a buffer manager, which keeps some blocks read from disk in memory.
3. Slices do not need to be 2-dimensional. It just happens to be the case here since the underlying cube has three dimensions.

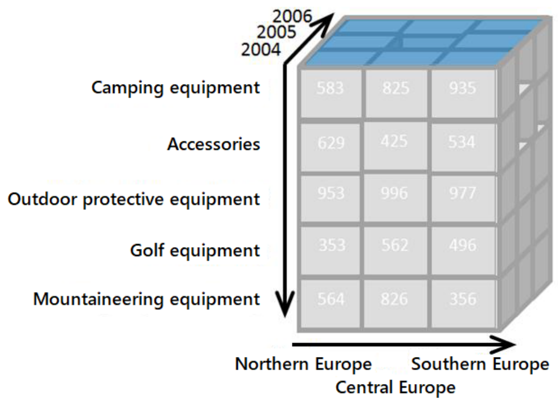


Figure 3.2: Illustration of a datacube representing a sales database.
Image is edited version of this, with this origial licence.

set as a cell has to exist for every possible combination, regardless of if all combinations exist in the underlying dataset[4]. For example if there were no sales in 2005 in the dataset illustrated in Fig. 3.2, one third of the space the cube occupies is effectively wasted.

3.2 Crossfilter

Crossfilter[15] is a JavaScript library for making interactive coordinated views in the browser. It is an entirely client-side library, meaning the entire dataset is transferred to the client on load. The library supports many popular visualizations, such as bar charts, line charts, tables, and scatter plots.

Up to a certain amount of data and dimensions, it is possible to make very fast interactive visualizations with Crossfilter ($<$ 30ms per interaction). One of its most attractive features is the possibility of creating custom grouping, aggregation, and filtering functions, which means that the axes of charts and the way charts are coordinated can be modified to suit even the most outlying needs.

Crossfilter achieves its speed by making use of the assumption that most explorative actions are incremental. Filtering and aggregation is done on smaller and smaller subsets of the dataset as more filtering operations are done. This keeps the interaction latency low and decreases until a filter gets reset and the active subset increases. To achieve this, Crossfilter relies on sorted indexes and

4. Simplification. Sparse representations do exist, but come at a cost for lookups.

a few optimizations using bit-wise operations. Crossfilter scales up to datasets with a few million data points, but performance drops of quickly with either volume or number of filterable dimensions. Cold-start latency scales with size and the number of dimensions as data must be transmit to the client, the client must decode the data, and the client to compute the indexes. The server for a Crossfilter-application can be static and serve the data as JSON or CSV, while the application itself is served as JavaScript-, HTML-, and CSS-files.

3.3 Nanocubes

A Nanocube is a data structure tackling the often prohibitively large size of data cubes. Nanocubes support queries to visualize spatiotemporal data in charts and heatmaps. The query responses are singular aggregates, binned aggregates, and tiles of heatmaps to be overlaid on a world map. The queries are evaluated very quickly (generally <10ms, even on a laptop) but are constrained by network latency and bandwidth. It is also not possible to query a nanocube for individual records, as the data structure only stores aggregates.

To achieve their speed, nanocubes use their own tree structure to index the data. As a query traverses the tree, it will first visit spatial nodes, which form a quadtree. Thereafter, it traverses nodes branching based on categories. Finally, in the leaf nodes are time-series stored as sparse prefix sum arrays.

The time and space complexity of building a nanocube is heavily impacted by the dimensionality and branching factors of the categories, as well as volume. This is illustrated in their paper by some of their example datasets and cubes: One nanocube with 210M records took 6 hours to build and uses 46GB, while another with 1B records took 4 hours to build and occupies 4.5MB.

3.4 imMens

imMens[13] is a system for creating interactive visualizations in web-browsers. It is built around the principle that interactive visualizations should scale with the resolution of the visualizations, and not the number of records in the dataset. imMens supports visualizations of aggregated bins in terms of bar charts and gridded heatmaps, where the aggregate function is *count*.

Gridded heatmaps are prioritized in imMens, and the data structure supporting the systems real-time interactivity is built around them. imMens is built by projecting data cubes onto 2-d *data tiles* which are encoded as PNG-images which are read using WebGL[5] in the browser. As data cubes become unwieldy when there are many bins, each tile represents a sub-square of the data holding area of the heatmap. Furthermore, to be able to project the data cube source into its 2-d tiles, imMens is limited to use 4 dimensions.

Using WebGL to handle both filtering and rendering, imMens achieves brushing performance of approximately 50hz and up on consumer PCs, even for datasets with billions of records. imMens does not support cross-filtering, only one filter can be active at a time. The data tiles map from each chart to the others, and would have to be recreated after each filter action to be able to compound them.

3.5 Falcon

Falcon[14] is another system for making CMV data analysis applications, and can be seen as a continuation of the imMens project. Similarly to imMens, Falcon is built to scale - in terms of latency on interactions - with the resolution of its visualizations rather than the amount of underlying records. Falcon continues the idea of data tiles, but with changes to what they comprise. There is no native support for heatmaps in Falcon, but it adds support for cross-filtering. Supported visualizations in Falcon are bar charts and scatter-plots, with *count* as the only supported aggregate function.

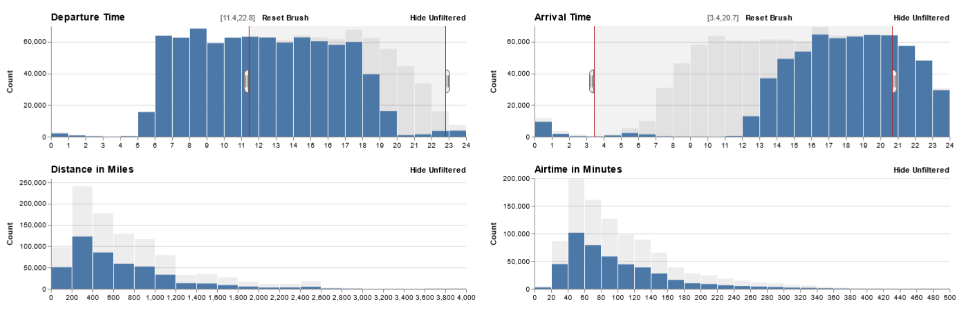


Figure 3.3: Example application made with Falcon. Source: `https://vega.github.io/falcon/flights/`

Falcon introduces the concept of *active* and *passive* views. An application has one active view at a time, while the remaining views are passive. The active view is the view the user is currently interacting with, i.e. brushing. A data tile in falcon is able to compute all possible states of one view based on the filtering in another. For each passive view, there is a tile based on the active view. Since

5. WebGL is a subset of OpenGL, enabling graphics computation in web browsers. WebGL is mostly used for rendering, but can also be used for GPGPU purposes.

Falcon supports cross-filtering, the tiles have to be recomputed every time there is a new active view. To do this quickly, Falcon uses the GPU-accelerated database OmniSciDB[6] in a back-end service which serves the new tiles to the front-end client.

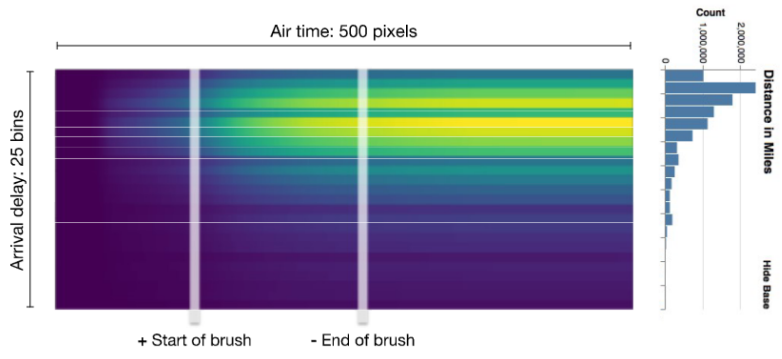


Figure 3.4: Illustration of a Falcon data tile, borrowed from the falcon paper[14]. In this example, *Air Time* is the active view, and *Arrival delay* is the passive view.

In Figure 3.4, we can see how a data tile maps filtering done in an active view to the bins of a passive view. The with of tile is equal to the width of the active view in pixels, while the height is the number of bins in the passive view. Each row is prefix sum array of counts for a bin in the passive view (illustrated as the height of bars in this example). Each column is a bin in the active view, one bin for each pixel. This enables brush filtering to be done at the resolution of the screen used and the updates to passive views are done with a single operation per bin: The value at the index of **end-of-brush** minus **start-of-brush**. When this operation is done for each row in each tile, all the bins for the passive views have been calculated. This operation is done in constant time if the number of bins is fixed, having a time complexity of $O(1)$. While this means that any range selection can be computed in constant time, it comes with the limitation that only brush selections are supported. If every other bin were to be selected, it would be twice as costly to sum the aggregates than if they were not stored as cumulative sums.

3.6 Summary

Databases, data cubes, and nanocubes can meet most of the filtering and aggregation needs for a framework of the kind we want. They are all however, limited by network connections on each user action and can be expensive

6. `https://www.omnisci.com/platform/omniscidb`

to compute and store. If it was feasible to send entire data cubes to the clients, it would make sense to build a CMV framework around them. The size of data cubes and the unpredictable size of nanocubes however, makes this unpractical.

Crossfilter shows us a solution where the entire dataset is sent to each client without requiring any pre-processing. This way, they remove the network impact on latency on user actions, but it severely limits the of data that can be used. imMens on the other hand, does pre-process the data which increases the supported data size. They do this however by trading away cross-filtering - it supports only one active filter at a time.

Falcon represents a middleway for all the other solutions by handling filtering with data on the client as well as by pre-processing new data tiles continually on the server. Falcon supports large dataset sizes, cross-filtering, and has low latency on most user interactions. These favorable qualities come with some draw-backs: Falcon only supports count as its reduce function, only supports range filters, only supports continuous dimensions, and it is built around a GPU-accelerated back-end. Falcon's performance and biggest draw-back stems from the design of its data tiles. By using prefix sums to facilitate fast filtering, they are not only limited to brush selections, but also to visualizations where the bins are are places next to each other in a fixed order. Making for example a filterable choropleth map almost useless as polygons would have to line up to be able to be selected.

4 Design

In this chapter we present the requirements for Slicer and its design. Slicer is a framework consisting of multiple components, with differing requirements so that each component encapsulates specific functionality. We will present their architecture, the high level design and the intra- and intercommunication required for performant CMV given limitations in web-browsers and commodity computers. The design is centered around methods to facilitate the requisites for CMV, with real-time performance constraints.

Slicer's architecture consists of two separate but communicating systems, a front-end JavaScript library and a back-end API. Here we will present the whole system's design, but emphasizing aspects enabling Slicer's performance and features.

Deciding on the fundamental design approach for Slicer has made up the principle part of this work in terms of time used. After reviewing the related literature and experimenting with various approaches (See Section 7.4), it became apparent that a tile-based approach was most tenable. Instead of competing with the culmination of approximately 10 years of research[1], Slicer is inspired by it.

1. imMens and Falcon were developed successively, largely by the same group.

4.1 Requirements

Slicer's requirements derive from the context outlined in Section 1.5 and findings in the literature regarding latency sensitivity. Here we outline both functional and non-functional requirements.

4.1.1 Non-functional requirements

- **Performance**
  - Latency sensitive operations should scale with visualization resolution.
  - Latency sensitive operations should be performed near 60Hz on commodity laptops.
  - Latency insensitive operations should take less than 500ms.
  - Performance goals should hold for datasets up to at least 20 million records.
- **Flexibility**
  - Slicer applications should be easy to create and run on commodity hardware.
  - Slicer visualizations should be able to be build into existing websites.
- **Extendability**
  - Slicer should facilitate addition of types of views and filters.
  - Additional aggregate functions should be possible to add in the future.

4.1.2 Functional requirements

- Must be able to use CSV-files as data sources.
- Must support visualization of relational data in binned aggregates.

- Must support linking and brushing.
- Must support single select.
- Visualizations must include a meaningful way of displaying geographical data.

4.2 User Interfaces

To achieve its requirements, Slicer is designed to support visualizing binned aggregates in bar charts and choropleths. Displaying geographical with choropleths avoids ove plotting and provides more meaningful bins than uniformly gridded heatmaps. The views can be interacted with by a user brushing or clicking to select bars or choropleth polygons. Selections can be active in multiple views at a time, thus enabling cross-filtering. The height of bars or color intensity of polygons is used to indicate the value of the aggregates for each bin. Hovering bins shows a tool-tip with exact values. An example application with a map and several bar charts can be seen in Figure 4.1.

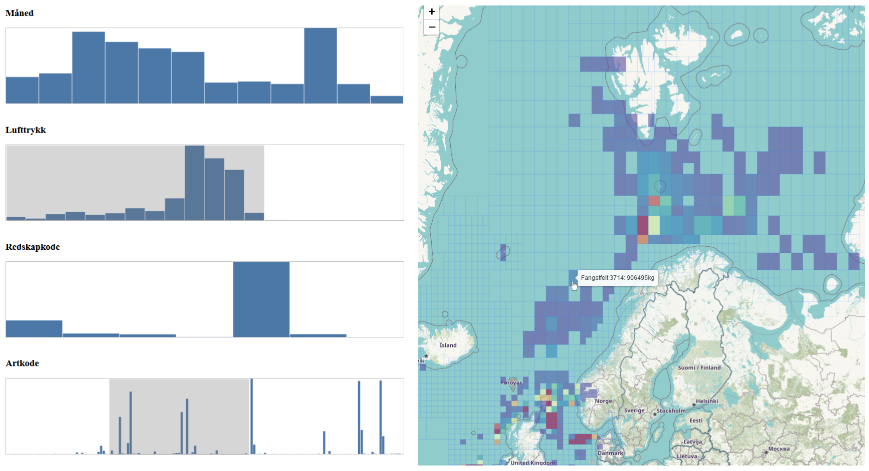


Figure 4.1: Example Slicer application.
Note: this screenshot is from before labels on chart axes has been added.

The geometry of the choropleth features has to be provided by the developer in a GeoJSON-file. GeoJSON is a standardized format for storing geographical data, taking the form of a JSON-file with a strict schema[21]. If a GeoJSON-file is not provided with the dataset, many preexisting GeoJSON-files are freely available online[2], covering common use cases such as countries, states, or

2. See for example: https://geojson-maps.ash.ms/ and https://eric.clst.org/tech/usgeojson/

cities.

4.3 A tile-based approach

The design of Slicer is inspired by the idea that visual analysis should scale with the resolution of the visualizations and the observation made in [11] that some user actions are more latency sensitive than others. Slicer handles latency sensitive operations efficiently in the client powered by data structures provided by the server. The concept of *active* and *passive* views from earlier work is continued. Some latency when switching active views is accepted in trade for real-time filtering. Slicer also continues the use of data tiles to enable filtering and coordination between active and passive views, although with a novel design, use, and computation.

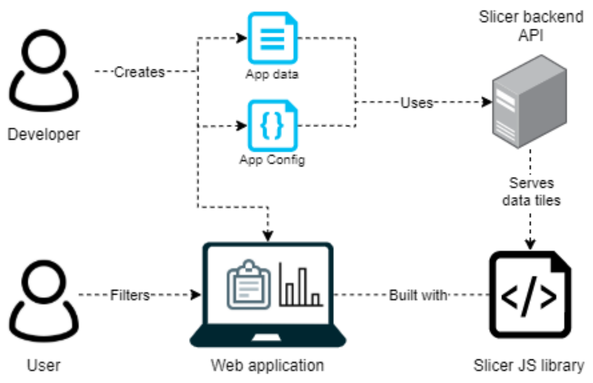


Figure 4.2: Birds eye view of a Slicer application.

In Figure 4.2, we see how the data tile approach is designed to fit into Slicer applications. Developers create the client application and include the Slicer JavaScript library, then they provide the configuration and data files to the Slicer back-end. This is all the back-end needs to prepare for serving data tiles and when a user hovers a view - *activating* it - in the application, the back-end creates and serves the necessary tiles to enable filtering.

Slicer's back-end relies on a in-memory columnar database to quickly compute data tiles. This database is set up on the first start of new Slicer application. Indexes are built and extra columns are added to speed up filtering and grouping, as outlined in Section 4.5. Upon building the database, the initial unfiltered bins for all views are computed and cached. These are served when a client connects so that it can see all views immediately. The back-ends life-cycle consists mainly of waiting for requests, as illustrated in Figure 4.3.

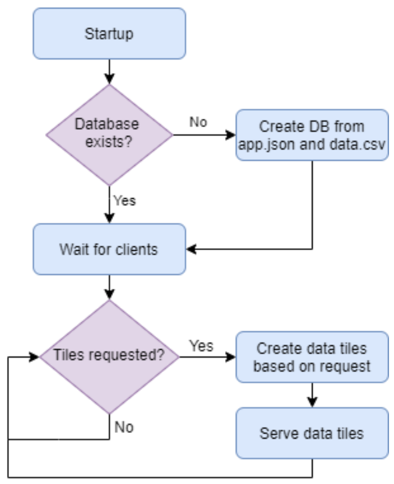
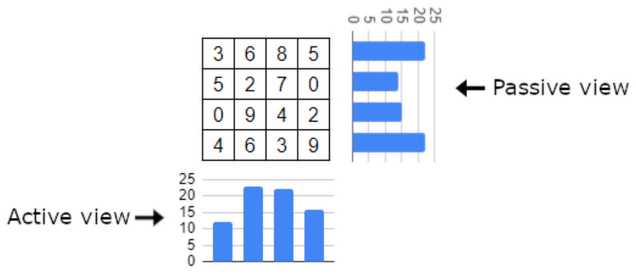


Figure 4.3: Behavior flowchart for Slicer back-end.

4.4 Slicer's data tiles

A Slicer data tile holds the information needed to compute bin values in a passive view for all possible selection combinations done in the active view. Since Slicer supports single-select, any combination of bins can be selected in the active view, meaning that a data tile holds the information for 2^b passive view bin configurations while using the space of b bin configurations, where b is the number of bins in the active view. There has to exist a tile for each passive view, so handling 2^b bin configurations for each passive view would quickly become unwieldy. Data tiles are created every time a new active view is set. Based on filters provided by a client, the server creates tiles for all the passive views. In an application with five views, the server would send four tiles, as the fifth view would be active one.



3	6	8	5
5	2	7	0
0	9	4	2
4	6	3	9

Figure 4.4: Example data tile. Column sums are equal to the bin values in the active chart and row sums are equal to the bins of the passive view. Here the two views are illustrated as bar charts.

Slicer data tiles are 2-dimensional, where the width is set by the number of bins in the active view and the height is set by the number of bins in the passive view. Each cell in the tile represents the aggregate of values belonging to bin x in the active view and bin y in the passive view. The sum of each row[3] represents the bin value for a single bin in the passive view. An example of a tile is seen in Figure 4.4. In the example both views have four bins. When bins are selected in the active view, filtering the dataset, the bins for the passive view are updated to reflect only the sums of selected columns.

4.4.1 Using data tiles

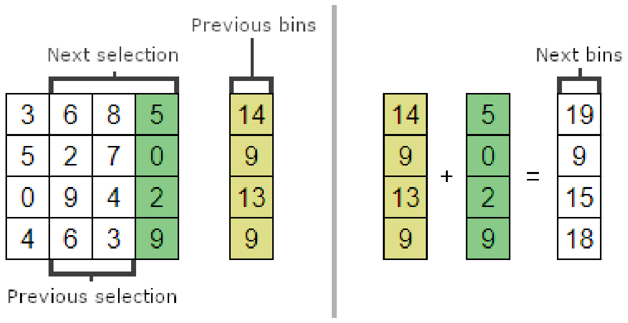


Figure 4.5: Illustration of how Slicer incrementally calculates the next bin values when filtering. The *delta* is highlighted in green here and the previous bins is highlighted in yellow. The values in *previous bins* is the sum of the columns from the *previous selection*

A naive algorithm to calculate new bin values when active views are filtered would iterate through all the selected columns in the data tiles and sum them together. in an application with a fixed number of total bins, such an algorithm would have a time complexity of $O(s_b)$, where s_b is the number of selected bins in the active chart (i.e. columns in the data tiles). Exploiting the observation that most selections done by brushing or single-selecting is incremental and applying a technique inspired by video codecs, Slicer handles most filtering operations in constant time. When brushing in a bar chart bars are added to the selection one by one. Likewise, when clicking on a bar or a polygon in a map the selection adds or removes one bin at a time. In video codecs, delta encoding is a common technique[4] Using delta encoding, the next frame in a sequence is stored as the change from the previous one. Observing that the bins for the previous selection is known, the bins for the next selection can be calculated as the previous added with the delta. As most selections add or remove one bin at a time, this calculation becomes taking the previous

3. For some reduce functions, two tiles are needed to get the correct values. E.g. to get the correct averages in the passive view, the the averages have to be combined with a tile containing the weights for each average.
4. The underlying principle for the technique is *data differencing*.

bin values and adding or subtracting a single column from the data tile. This process is illustrated in Figure 4.5.

4.4.2 Assigning bins

Slicer's design supports categorical and quantitative data. For categorical dimensions, one bin is assigned for each unique value in the dimension. For example, if a dimension called "color" only contains data with the values "red", "green", and "blue", a view visualizing this dimension would have three bins.

Quantitative data can have unique values for each data point. To visualize this type of data in a meaningful way, the range from the minimum to the maximum value in dataset is divided into uniformly sized sub-ranges. These sub-ranges form the bins, and any data point with a value within the sub-range of a bin, belongs to that bin. The number of bins to create for a view with quantitative data has to be set in the configuration file.

4.4.3 Creating data tiles

Data tile are created using an in-memory columnar database. When Slicer's back-end receives a tile request, it includes which chart is the active one and which filters are active in the passive views. Translating the filters to SQL, the database is queried once for each passive view. Each of the responses contain the data for a data tile.

To improve this process, another *delta differencing* technique is used. When a Slicer back-end starts, unfiltered data tiles for each view are computed and cached. This results in $v * (v - 1)$ cached tiles, where v is the number of views in the application. These tiles are then used later in the computation of new tiles when the back-end receives requests from clients. If the filters received are estimated to filter out more than 50% of the rows in the database, the filters are inverted before being translated into SQL. The resulting tiles are then subtracted from the cached tiles. Doing this ensures that the database never computes aggregates on more than 50% of its rows. Computing aggregates is the slowest operation done in the queries, so the resulting improvement effectively halves the worst case complexity of computing data tiles.

4.5 Database design

The database schema for a Slicer application is decided by the dataset and configuration file provided. All fields in the CSV-file which either is the dimension for a view or is used in the aggregate function for a view, are added as columns in the database. Additionally, there is added one new column for each view, containing the bin id for each row. These bin ids indicate which bin a row belongs to in the respective views, and are essentially the index in bin array for each view.

The columns with the bin ids are used for filtering and grouping. Since all filtering in Slicer is either a range of bin ids or a collection of bin ids, these can be used instead of the actual underlying data. To speed up filtering a regular nonclustered index is created on each of them. When data tiles are computed, the bins are grouped by bin id in both the active and the passive view. To speed up the grouping, a multifield index is created for each possible combination of two views. N choose 2, $\binom{N}{2}$, where N is the number of views in the application, such indexes are created.

4.6 Architecture

Slicer applications consist of a client web-application and a server. Both the client and server software consists of multiple components each. The architecture is set up to facilitate efficient communication and data transfer between the clients and the server. To reduce latency, bandwidth, and server load, the architecture is a client-server architecture with a fat client[5]. While the server computes new data tiles when a new active view is set, all the computation for real-time filtering is done by the client. A simplified overview of the logical components of Slicer can be seen in Figure 4.6. The implementation has more classes and interfaces, but these components illustrate the main distribution of computation and communication done.

Communication between clients and server is done with two protocols: one with low overhead per request for messages, and another for data transfer. Messages are sent over the WebSocket protocol[22], while data tiles are transferred over HTTP. For messages, such as requests for new tiles, it is important that the server can handle them as quickly as possible; Before the request is received the server cannot start making new data tiles. Another aspect is server load. There are many more messages sent than data tiles, and WebSockets scales better

5. Could also be called edge computing, where the concept is to keep computation near the user with the same performance goals as a fat client.

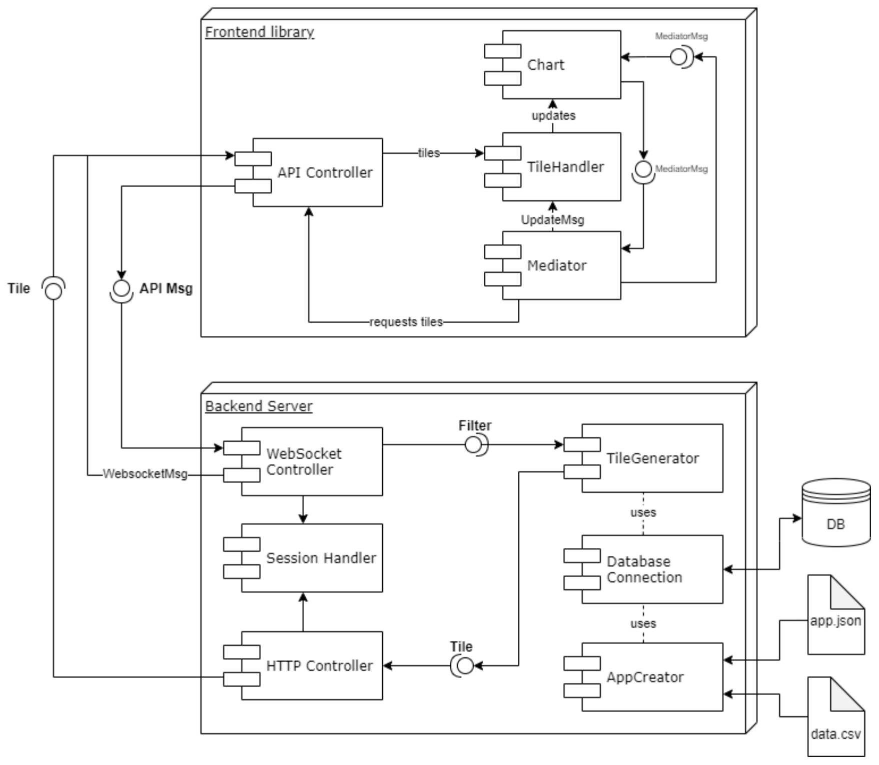


Figure 4.6: Logical component diagram of Slicer.

than HTTP requests for concurrent messages[23]. By using HTTP for data tiles, the tiles can be sent concurrently while keeping the WebSocket connection open for other messages.

Slicer's front-end code consists of loosely coupled components which communicate through a mediator[6]. Each component can subscribe to specific topics and can send messages with specific topics. This way the different view can be coordinated without knowing about each other, simplifying the code and making it easy to implement new views in the future.

6. This design pattern is sometimes also called Pub/Sub and is similar to an Event Bus.

5 Implementation

In this chapter we present the implementation of Slicer. The implementation includes a Web-API and a JavaScript library, containing many necessary components surrounding and facilitating Slicer's features and performance. Here, we will briefly cover supporting components, and keep the primary focus on components directly tied to Slicer's design.

The code for the implementation is found in the `source_code.zip` archive.

5.1 Programming languages

To be able to use an iterative approach and validate ideas quickly, Python[24] was chosen as the language for implementing the back-end. The authors fluency in the language and its rich selection of packages, made it a natural choice when developer time was a primary concern. Packages such as numpy[25] and pandas[26] makes data processing operations easy to implement and reasonably fast as they are implemented in c[27]. The FastAPI package[28] ensured that setting up a web API with WebSocket functionality was trivial, allowing for most time to be spent on experimenting with algorithms and data structures.

Slicer's front-end code, responsible for handling data tiles, user input and rendering visualizations is implemented in TypeScript[29], which is a language that transpiles to JavaScript. The alternative would have been to use JavaScript directly. This choice is primarily done due to preference, as making the code extendable was desirable and in the authors experience it is more comfortable to extend upon typed code.

5.2 Front-end

The implementation of Slicer's front-end builds on principles and techniques outlined in **Clean Code**[30] and **Design Patterns**[31] for writing reusable and extendable code. Namely, the principles of "*single responsibility*", "*composition over inheritance*", and "*relying on abstractions, not implementations*" are applied. The application of these principles decouples internal components and makes the implementation both easier to understand and extend upon in the future. TypeScripts support for types, interfaces, and abstract classes, made the techniques uncomplicated to adopt.

5.2.1 Mediator

The central component of Slicer's front-end is the mediator. Any part of the front-end code can subscribe to messages or send messages to the mediator. There always exists a mediator, and there only exists one as it is implemented as a singleton. The mediator itself is not exposed, only its `subscribe()` and `send` methods are. All messages have to be one of the `MediatorMsg` types, which can be seen in Appendix C. These messages have a `Subject`, and when subscribing, objects can choose to only listen for messages on certain subjects. The mediators implementation can be seen in `slicer-frontend/src/Mediator.ts`.

5.2.2 API-connector

The API-connector is what connects clients to the back-end server. It has methods for requesting metadata for views and tiles, sending data tile requests, and downloading new data tiles. Messages are sent via a WebSocket connection while data tiles are downloaded using concurrent HTTP requests.

The API-connectors implementation can be seen in `src/WebSocketAPI.ts`

5.2.3 Views

Slicer's implementation currently supports two types of views: Bar charts and choropleth maps. All views keep their bin values in a typed JavaScript array. These arrays are used when rendering bins, and are updated when a users filters. The connection between bin arrays and views is shown in Figure 5.1, where it can be seen how the bin values determine the height of the bars in a bar chart.

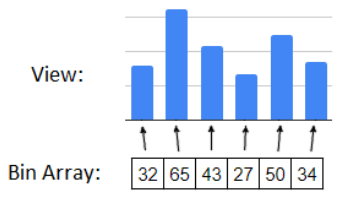


Figure 5.1: Connection between bins and views. Here exemplified with a bar chart.

Bar chart

Bar charts are implemented with HTML-Canvas elements[32]. Bar charts automatically fill their parent DIV-element when created, making customization of placement and size a matter of editing CSS for the developer.

To render the bars of a bar chart, the method `BarChart.render()` is called. This method iterates the charts bin values and draws a rectangle for each. The height is calculated as the percentage of the maximum bin value. The implementation of `BarChart.render()` is found in the file `slicer-frontend/src/BarChart.ts` and in Listing 5.1.

Code Listing 5.1: BarChart.render() method.

```
render(): void {
    this.context.clearRect(0, 0, this.canvas.width, this.canvas.height);
    this.rs.bins.forEach((val, i) => {
        const x = this.rs.bar_width * i + i * this.rs.bar_spacing;
        const h = (val / this.rs.max_bar_value) * this.canvas.height;
        const y = this.canvas.height - h;
        this.context.fillRect(x, y, this.rs.bar_width, h);
    });
}
```

Choropleth

Choropleth layers are implemented using the leaflet[1] library. Leaflet can draw regular maps and has native support for rendering GeoJSON. This feature and leaflets lightweight footprint (39kB of bundled JavaScript), made it the choice for map rendering.

The initial polygons are created using leaflets built in `geoJSON()`-method. For each polygon an onClick-event is added so that clicking them trigger a filtering operation. This event makes use of a key lookup which maps the "key"-property of the GeoJSON features to a bin index. The lookup is a part of the AppConfig-object the client receives upon connecting to the back-end, of which the full implementation can be seen in Appendix D or `slicer-frontend/src/Types.ts`.

As with the bar charts, the choropoleth layer updates when a filtering happens. To reflect the changes made by a filtering operation, each polygon gets a new color based on the updated bin values. To set this color, the chroma-js library[2] is used. The implementation of the rendering is seen in Listing 5.2, and the full implementation for GeoJsonLayers can be seen in the file `slicer-frontend/src/GeoJsonLayer.ts`.

Code Listing 5.2: GeoJsonLayer.updateView() method.

```
private updateView(): void {
    this.group.eachLayer((d: any) => {
        const idx = this.config.lookup[d.feature.properties[this.key]];
        const val = this.rs.values[idx];
        const r = val / this.rs.max_value;
        const i = val > 0 ? true : false;
        const o = val > 0 ? 0.6 : 0.0;
        d.setStyle({
            fillColor: this.colorscale(r).hex(),
            interactive: i,
            fillOpacity: o });
    });
}
```

5.2.4 Tile Handler

Filtering is the main feature of Slicer and is implemented with its data tiles. Data tiles are handled by the TileHandler, which is implemented in

1. `https://leafletjs.com/`
2. `https://gka.github.io/chroma.js/`

`slicer-frontend/src/TileHandler.ts`. Notified by its mediator subscriptions, the tile handler fetches new tiles when a new active view is set and updates bin values when filters change.

To fetch data tiles, the tile handler uses the API-connector and JavaScript promises[33] to establish concurrent downloads as seen in Listing 5.3. For small downloads, the HTTP-handshakes[34] make up a large portion of the time. Data tiles are small, often less than 1kB in size. Thus it is important to ensure that the time taken by each handshake overlaps.

Code Listing 5.3: Fetching Data Tiles.

```
private async setTiles(configs: TileConfig[]): Promise<void> {
    const promises = configs.map(c => this.api.getTileData(c.name));
    Promise.all(promises)
        .then(data => {
            configs.forEach((c, i) =>
                this.tiles[c.name] = { config: c, data: data[i] }
            );
        });
}
```

When a view is filtered, it sends a Filter-message to the mediator. Slicer differentiates between two types of filters. One type is the Range-filter, which has a start bin and a stop bin. Range-filters are the result of a brush selection. The other type of filter is the Category-filter, which contains a list of bin indexed which are selected. Category-filters are the result of single-selecting. In Appendix E, the definition of the filter types can be seen. The tile handler is subscribed to these messages, and calculates the delta from the previous filter when received. The delta is then used when updating bin values.

Deltas are stored as two arrays: One for added bins and one for removed bins. Calculating the delta for a Range-filter is done by determining the difference in the start and end bins for the selection. Bins between the previous selection and the current one are added to the *added* or *removed* array, depending on whether they are in the current selection or not. Deltas between Category-filters are calculated by looking at the intersection between the previous selected bins and the current. This is implemented in the code shown in Listing 5.4.

Code Listing 5.4: Computing filter delta.

```
if (previous.type == FilterType.Range
    && current.type == FilterType.Range) {
    const additions = [];
    const removals = [];

    if (previous.range[0] < current.range[0])
        removals.push(...getRange(previous.range[0], current.range[0]));
    else if (previous.range[0] > current.range[0])
        additions.push(...getRange(current.range[0], previous.range[0]));
    if (previous.range[1] < current.range[1])
        additions.push(...getRange(previous.range[1], current.range[1], true));
    else if (previous.range[1] > current.range[1])
        removals.push(...getRange(current.range[1], previous.range[1], true));

    if (!additions.length || !removals.length)
        this.delta = false;
    else
        this.delta = [additions, removals];
}
else if (previous.type == FilterType.Categorical
         && current.type == FilterType.Categorical) {
    const additions = current.categories.filter(x =>
        !previous.categories.includes(x));
    const removals = previous.categories.filter(x =>
        !current.categories.includes(x));
    this.delta =  [additions, removals];
}
```

Updating the bins of a view is done by iterating the delta arrays. For each bin index, each value on the y-axis of the data tile is added to or removed from the corresponding view bin. Data tiles are implemented as flat typed JavaScript arrays. Accessing a value from an x and a y coordinate therefore requires knowledge of the dimensions of the data tile. The tile handler has this knowledge and specific indexes in data tiles are calculated as: $idx = x + y * w$. The code iterating the delta and updating a views bins is shown in Listing 5.5.

Code Listing 5.5: Updating bins using data tiles and deltas.

```
for (let x of additions) {
    for (let y = 0; y < h; y++)
        bins[y] += tile.data[x + y * w];
}
for (let x of removals) {
    for (let y = 0; y < h; y++)
        bins[y] -= tile.data[x + y * w];
}
```

5.3 Back-end

5.3.1 Initialization

The primary task of Slicer's back-end is to compute data tiles based on filtering done client-side quickly. To do so, the life cycle of the back-end consists of two parts: Initialization and runtime. The initialization prepares metadata and data structures to be used at runtime serving clients.

During the initialization phase, the supplied configuration file and dataset are processed. The phase consists of 5 primary steps:

- Compute bin information for each view; The category or value intervals, the represented dimensions, and which reduce function are among the information needed.

- Create a database based on the previous step; Reading data from CSV into the database and adding indexes and columns for bins. Compute initial bin values for each view, so these can be sent immediately to connecting clients.

- Compute initial bin values for each view, so these can be sent immediately to connecting clients.

- Compute unfiltered data tiles for each view. These are sent to clients when a new active view is set without any present filters and used to compute data tiles when >50% of the dataset is selected.

- Compute histograms for each view. These are similar to the initial bin values but only use count for reduction. Using histograms can estimate if >50% of the dataset is selected - and thus if the cached data tiles should be used when computing new tiles.

5.3.2 Communication

Communication between clients and the back-end is enabled by the FastAPI package[28]. Endpoints for data, metadata, and WebSocket communication are set up as seen in `slicer-backend/slicer/main.py`. The endpoints use the GET path[34] to determine which data to respond with. For example, the endpoint for data tiles seen in Listing 5.6 shows how application name, WebSocket session id, and view names are used to return data tiles.

Code Listing 5.6: Data tile endpoint.

```python
@api.get("/{app_name}/{session_id}/tile/{view_name}")
async def get_tile(app_name: str, session_id: int, view_name: str):
    app = app_lookup[app_name]
    data = io.BytesIO(app.get_tile(session_id, view_name).tobytes())
    return StreamingResponse(
        content=data,
        media_type="application/octet-stream"
    )
```

The API sends two types of responses: JSON and octet-streams (raw bytes). When the front-end will use the message as JavaScript objects internally, JSON responses are used. Consequently, all messages except those sending data tiles and initial bin values are JSON. These messages are encoded as text and have to be decoded and parsed client-side[3]. Therefore, data responses are sent as byte arrays, avoiding overhead; the server avoids extra data encoding, and the clients avoid decoding and parsing. The responses are also smaller, as encoding numbers as text is space-inefficient.

5.3.3 Database

The columnar database used by Slicer is DuckDB[35]. DuckDB instances are in-process and can keep data in-memory. The in-process nature removes any need for setting up a separate database before using Slicer. As long as the duckdb python package[4] can be installed, Slicer will create the database itself. Being in-process, DuckDB gets a performance benefit since it can use intra-process Local Procedure Calls (LPC), which are faster than alternative Remote Procedure Calls (RPC)[5].

5.3.4 Creating data tiles

Slicer relies on its database to compute data tiles. Clients requesting tiles send filters and the name of their active view to the server. This information is then translated into several SQL-queries, one for each passive view. An example of such a query is seen in Listing 5.7 and the code for creating the SQL queries can be seen in Appendix A. The queries result in collections of tuples on the form (*x, aggregate, y*), where grouping is done on *x* and *y*, ensuring a single

3. `https://tc39.es/ecma262/multipage/structured-data.html#sec-json.parse`
4. https://pypi.org/project/duckdb/
5. See: https://raima.com/database-terminology/

tuple for each combination. The value in *aggregate* represents the aggregate of values in the filtered dataset belonging to bin *x* in the active view and bin *y* in the passive view.

Code Listing 5.7: SQL-query used to create data tiles. This example is for a dataset of fishing catches. This specific SQL query creates a tile from an active view displaying months to a passive view displaying species.

```sql
SELECT month_bin, SUM(CatchWeight), species_bin
FROM data
WHERE (year_bin BETWEEN 2 AND 14) AND (tool_bin IN (3,6,9))
GROUP BY month_bin, species_bin
```

Since Slicer clients do not expect collections of tuples as data tiles, so they have to be converted. Both Slicer front-end and back-end stores data tiles in typed arrays, which python does not support natively. Therefore the NumPy[25] package, which has an interface for creating arrays in C, is used. The first step to convert a set of tuples is to create an empty array big enough to store the tile, hence the length is set to the product of number of bins in the active and passive view. Then the tuples are iterated and the *x* and *y* values are used to find the respective index in the array, and the *aggregate* value is inserted at that index. This code implementing this process is shown in Listing 5.8.

Code Listing 5.8: Code for turning SQL result into data tile.

```python
def result_to_nparray(result, from_view, to_view):
    w, h = from_view.nbuckets, to_view.nbuckets
    data = np.zeros(w * h, np.uint32)

    # References to avoid hash table lookups inside the loop
    x = result['x']
    y = result['y']
    v = result['v']

    for r in range(x.size):
        data[x[r] + (y[r] * w)] = v[r]
    return data
```

Using cached tiles

To improve response times, cached tiles are used. If a request without filters is received, the server responds with cached tiles immediately. If a request where the filters are assumed to filter out less than 50% of dataset is received, the cached tiles are used in the computation of new tiles.

Estimating the rows selected by the active filters is done by applying the filters to the histograms mentioned in Section 5.3.1. For each filter and each histogram, the counts in the selected bins are summed. These sums represent the number of selected rows in the database, if the respective filter was the only one.

If the most restrictive filter results in a row count of less than half row in the database, we know for sure that more than half of the dataset is filtered out. However, if the most restrictive filter results in a row count bigger than half the total row count, we assume that less than half of the dataset is filtered out. The code for estimating row count is shown in Appendix F.

When it is assumed that less than half of the dataset is filtered out, filters are inverted before converting them to SQL. This inversion results in very few SQL-queries where more 50% of the dataset is visited. After the tiles are computed with the inverted SQL, they are subtracted from the unfiltered cached tiles. Consequently, if a user has selected 90% of the dataset, only 10% is used to compute the new tiles.

5.4 Creating an application

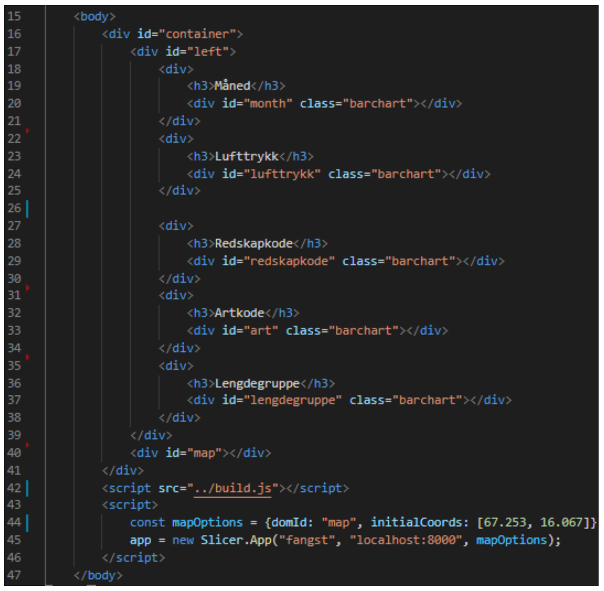
The process of creating an application with Slicer is designed to require minimal effort by developers. For the backend part of Slicer to work, a configuration file and CSV-file with data needs to be provided. If a map with a choropleth overlay is to be used, a GeoJSON-file containing the related polygons also has to be provided.

The configuration file has to be a JSON-file similar to the one seen to the right in Figure 5.2. It has to contain the location of the data and the descriptions of the views for the application. The descriptions depend on the type of view, but all include a name for the view. For bar charts it has to state if the data on the x-axis is categorical or quantitative, which dimension to use for the y-axis, and which aggregate function to use. Map layers (choropleth), need to state the key linking each data point to a polygon in the GeoJSON, which dimension decides intensity, and which aggregate function to use.

The front-end is created as a regular web-site which imports the Slicer JavaScript library. To place views in the application, the developer creates DIVs[6] where the id[7] attributes correlate to the name set in the config file. Instantiating the Slicer components is done by passing in parameters for the URL of the backed

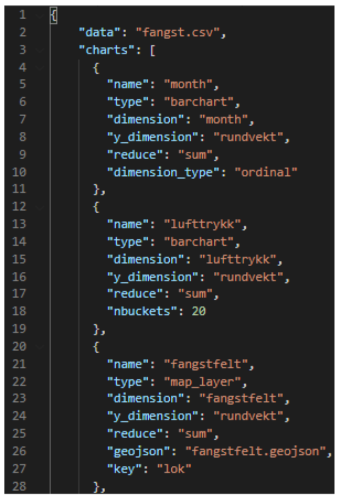
6. `https://www.w3schools.com/Tags/tag_div.asp`
7. `https://www.w3schools.com/htmL/html_id.asp`

server and the name of the application. The HTML and JavaScript code for the application shown in Figure 4.1, can be seen to the left in Figure 5.2



```html
<body>
    <div id="container">
        <div id="left">
            <div>
                <h3>Måned</h3>
                <div id="month" class="barchart"></div>
            </div>
            <div>
                <h3>Lufttrykk</h3>
                <div id="lufttrykk" class="barchart"></div>
            </div>

            <div>
                <h3>Redskapkode</h3>
                <div id="redskapkode" class="barchart"></div>
            </div>
            <div>
                <h3>Artkode</h3>
                <div id="art" class="barchart"></div>
            </div>
            <div>
                <h3>Lengdegruppe</h3>
                <div id="lengdegruppe" class="barchart"></div>
            </div>
        </div>
        <div id="map"></div>
    </div>
    <script src="../build.js"></script>
    <script>
        const mapOptions = {domId: "map", initialCoords: [67.253, 16.067]}
        app = new Slicer.App("fangst", "localhost:8000", mapOptions);
    </script>
</body>
```



```json
{
    "data": "fangst.csv",
    "charts": [
      {
        "name": "month",
        "type": "barchart",
        "dimension": "month",
        "y_dimension": "rundvekt",
        "reduce": "sum",
        "dimension_type": "ordinal"
      },
      {
        "name": "lufttrykk",
        "type": "barchart",
        "dimension": "lufttrykk",
        "y_dimension": "rundvekt",
        "reduce": "sum",
        "nbuckets": 20
      },
      {
        "name": "fangstfelt",
        "type": "map_layer",
        "dimension": "fangstfelt",
        "y_dimension": "rundvekt",
        "reduce": "sum",
        "geojson": "fangstfelt.geojson",
        "key": "lok"
      },
```

Figure 5.2: HTML and JS required for the example application (left) and some of the JSON configuration for the same application (right).

6 Evaluation

The primary focus of Slicer's design and implementation, has been to add features while preserving the performance of existing work. Slicer supports more aggregate functions, view types, data types, and selection types than earlier work with the same performance goals. In this chapter we investigate how adding these features has affected Slicer's performance.

There are two primary sources of potential latency in Slicer: Filtering and computing new data tiles. We will run benchmarks for each of these.

6.1 Experimental Setup

The Slicer application used in the benchmarks is the same as can been seen in Figure 4.1 (note: there is one more bar chart below "Artkode", which is not show in the screenshot for the figure). The dataset contains records of fishing hauls and has a total of approximately 20 million records. In the benchmarking application there are six different views with a total count of 1637 visible bins. These six views are:

1. **Fangstfelt** A choropleth layer with 1385 bins showing months with CatchWeight sum-aggregated to set the color intensities.

2. **Måned** A bar chart with 12 bins showing months with CatchWeight sum-

aggregated on the y-axis.

3. **Lufttrykk** A bar chart with 20 bins showing air pressure with CatchWeight sum-aggregated on the y-axis.

4. **Redskapkode** A bar chart with 7 bins showing fishing tool code with CatchWeight sum-aggregated on the y-axis.

5. **Artkode** A bar chart with 204 bins showing fishing tool code with CatchWeight sum-aggregated on the y-axis.

6. **Lengdegruppe** A bar chart with 9 bins showing fish length groups with CatchWeight sum-aggregated on the y-axis.

Two separate computers are used to do the different benchmarks. A laptop is used to benchmark filtering performance. While tile computation is benchmarked on a desktop computer. The specifications of these are listed in Table 6.1.

Laptop	
Model	Lenovo Ideapad 720S-14IKB
CPU	Intel i7-8550U @ 1.8GHz (4.0 max)
RAM	8GB DDR4 @ 2400
OS	Pop!_OS 20.04 (Linux)
Browser	Chromium 93
Desktop	
CPU	AMD Ryzen 5900x @ 4.5GHz
RAM	32GB DDR4 @ 3200MHz
OS	Windows 10 Education 20H2

Table 6.1: Specifications of computers used in benchmarks.

6.2 Benchmarks

6.2.1 Filtering

Procedure

To measure filtering performance, 250 filter operations were timed. These operations included brushing in all the bar charts and single selecting in choropleth map. Timing the operations was done by exploiting the fact that all filtering operations are propagated through the mediator. When callbacks for all listener to Filter messages have been called, the updating of all views is done. The implementation of the timing is seen in Listing 6.1. Here it can be seen how the mediator send method is modified to measure the time before and after a message is propagated, and if the message is a filter message, the time between measurements is stored the `this.timings` array. When 250 filter operations have been measured, the timings are printed to the console. In addition to these 250, additional operations were inspected separately by printing the timing after each one.

Code Listing 6.1: Mediator.send() modified to time filtering.

```
public send(msg: MediatorMsg): void {
    const t0 = performance.now();
    this.subscriptions[msg.subject].forEach((cb) =>
        cb(msg.content));
    const t1 = performance.now();

    if (msg.subject == MediatorSubject.Filter) {
        this.timings.push(t1 - t0);
        if (this.timings.length == 250)
            console.log(this.timings);
    }
}
```

Results

The resulting measurements can be seen in Figure 6.1, where they are plotted as regular and cumulative histograms. As the left part of the figure shows, most operations (232 out of 250) took less than 8 milliseconds. The cumulative histogram shows that 90% of operations took less than 6ms and 99% took less than 15ms. All the outliers, 1% of the operations, took less than 30ms.

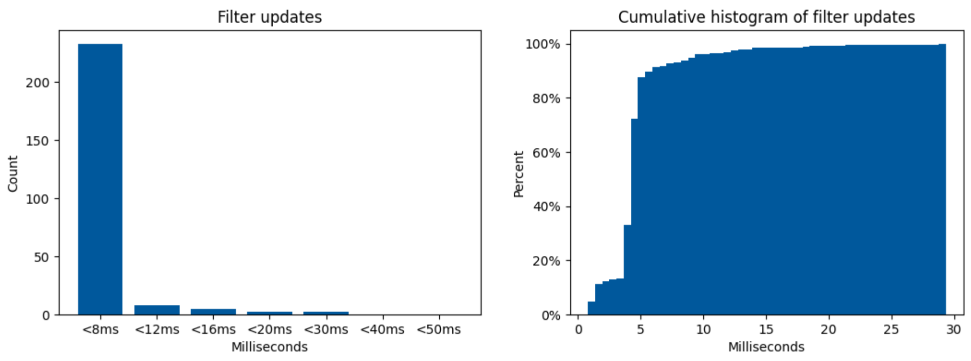


Figure 6.1: Regular (left) and cumulative (right) histograms of latency for 250 filter interactions.

Inspecting the timings of operations one by one made it possible to see differences in timings between the views. For all the bar charts there was no measurable difference in time taken to handle operations. The only persistent difference was seen when filtering in the chorolpleth, where the operations were much faster, averaging about 1ms v.s. 4ms in the other charts. This difference can be explained by the vast difference in bins between the choropleth view and the others. The choropleth has 1385 bins, while the others range from 7 to 204. As filtering a view updates all other views, but not itself, filtering in the choropleth leads to much fewer bin updates than filtering in any other view. The performance of a filter operation is determined by the number of bin updates it results in, which is reflected in the height of the data tiles.

We were not able to consistently reproduce outlier measurements, taking more than 15ms. There was found no relationship between view or type of filtering and outlier measurements. This observation leads to the assumption that these measurements are results of external factors affecting the resources available to the browser. Such factors may be the result of for example: OS-tasks or CPU boosting.

6.2.2 Data tile computation

Procedure

Data tile computation was investigated by measuring time when computing tiles for different data sizes and tile resolutions. To do this, four subsets of the dataset were made. These subsets contained 1, 5, 10, 15, and 20 million records each. The application from Section 6.1 was made with each of these datasets.

In the application, six bars in the "Måned"-chart were selected, and then the "Lufttrykk"-chart was hovered. This triggers a computation of data tiles in the back-end where approximately 50% of the dataset is selected. The computation of these tiles was then measured using Pythons built in *time*[1] package. Both the total time, and the time for each tile was computed. The code with the measurement for all tiles can be seen in 6.2 and for each tile in Appendix B.

Code Listing 6.2: Tiler.set_tiles() method with timing.

```
def set_tiles(self, session_id, from_view, filters):
    t0 = time.time()
    [...]
```

1. https://docs.python.org/3/library/time.html

```
    for tile in self.tiles[session_id].values():
        tile.set_data(self.data_provider.get_tile(from_view, [..]))
    print("setting tiles took:", time.time()-t0)
```

Results

The result of the data tile benchmark can be seen in Figure 4.6. Excepting the jump from one to five million records, performance appears to scale close to linearly with data size.

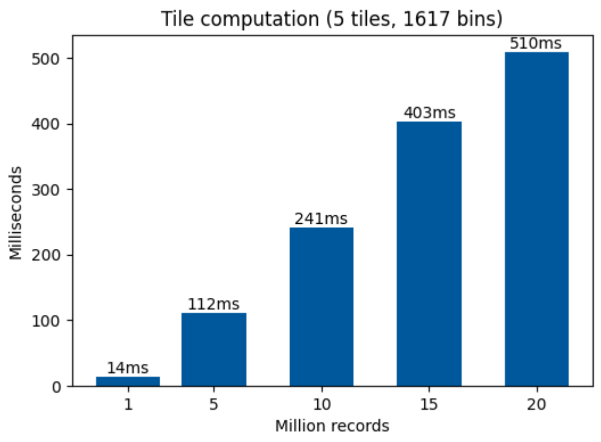


Figure 6.2: Performance measures of tile computation for the benchmark application. Measured with a 50% brush selection in "Måned" and "Lufttrykk" as the active view.

Table 6.2 shows measurements for each individual tile when the dataset had 20 million records. The figure shows that bin count affects tile computation performance as "fangstfelt" with its 1385 bins is by far the slowest tile to compute. This observation was reproducible and consistent. Interestingly, "art", which has 10 times as many bins as "lufttrykk", took shorter time to compute.

Data Tile	Milliseconds
fangstfelt	221
lengdegruppe	79
redskapkode	78
artkode	70
luttrykk	80
Total	528

Table 6.2: Detailed time taken when creating data tiles with a 20 million record dataset.

6.3 Analysis

The requirements in Section 4.1, state that filtering operations should be possible to do at 60Hz on a commodity laptop. To achieve 60Hz for continuous filtering, each operation has to take less than 16 milliseconds since $1s/60 = 16.6ms$. With 90% of the operations taking $< 6ms$ and 99% taking < 15, most operations were able to be done at above 160Hz, 100 frames per second more than the laptops display can handle. The next 9% of operations ran above 66Hz, while the outliers dipped to around 33Hz. The outlier measurements did not come in clusters however, so the dips usually lasted for a single frame at a time and were not perceivable in our experience.

Slicer was designed to scale with the resolution of the view, and not the dataset size. Therefore, the total number of bins in the application should decide its filtering performance. Here, we saw that views with high amounts bins were measurably slower to update than views with with fewer. This difference was only a couple of milliseconds at most, even when the amount of additional bins was more than 10x as many (e.g. "Måned" v.s. "Fangstfelt"). This small difference indicates that Slicer applications should be able to scale to much higher amounts of bins, while keeping most updates below 60Hz on a laptop.

According to the requirements, Slicer should be able to switch to a new active view in less than 500ms, while supporting at least datasets of 20 million records. This means that new data tiles should be computed faster than 500ms. Here, we have seen that Slicer almost meets this requirement - in the benchmark application at least. We also uncovered that the time it takes to compute a data tile not only depends on how many records it is built from, but also how many bins it has. In an application with six views with around 20 bins per view, we extrapolate from the benchmark numbers that Slicer would meet its requirements, even for dataset above 20 million records.

6.3.1 Comparison with Falcon

Falcon[14] is the natural project to compare Slicer's performance with. It was designed with similar goals and has commonalities in approach. The performance benchmark in Falcon's paper does not state the exact number of bins in the application used, but extrapolating from the illustrations, it is around 130, with 20-25 in each of its six views. In their graph showing filtering performance, it seen that that Falcon averages 50Hz in this application. This translates to 20ms per update. In contrast, Slicer's benchmark was done with more than 10 times as many bins and the same number of views with an average of 4.5ms per update. Falcon's benchmark was run on 3 year older hardware (2014 v.s. 2017 model laptop), which might accommodate for the performance difference.

Data tiles are also used in Falcon, and are computed in a similar way as in Slicer. In their paper, they state that they compute the data tiles[2] for their benchmark application in 130ms when the dataset has 7 million records. The most comparable benchmark we have done is with 5 million records, for which the same amount of tiles were computed in 112ms. In this comparison, our benchmark had 2 million fewer records, but approximately 1500 more bins. At these data sizes, the performances are close. However, Falcon scales better with size. They have a benchmark of the same application with 180M records in the dataset, where the tiles are computed in 1.7s. A difference in performance was expected as Falcon uses a GPU-accelerated database. Therefore it is surprising that Slicer is competitive for smaller datasets. Aggregate functions cannot be embarrassingly parallelized[3], hence the GPU-database is likely to have an overhead for combining calculations. This overhead does not necessarily scale with size, and might explain Falcon's favorable scaling compared with Slicer.

2. At visual bin resolution, not pixel resolution. Falcon does both, but this is most comparable to Slicer.
3. Even if a GPU has one core for each record, combining the counts or sums has to be synchronized.

7 Discussion

7.1 Geographical visualizations

Earlier works either do not support geographical visualization[14], or implements it using gridded heatmaps[13][36]. Gridded heatmaps create a fine mesh of the area with data points. This mesh consists of uniform cells (bins) filling the area, regardless of where the data points are. As a result there are created many bins, many of which often are empty. The performance of Slicer's tile based approach is tightly coupled with the amount of bins in the visualizations, making it undesirable to use gridded heatmaps. Instead, Slicer exploits the fact that the worlds geography has many natural divisions(bins). Using these natural sections are not only easier to understand, but will often offer a more meaningful way of filtering. In a heatmap, selecting an area is often done as a box select, which makes it virtually impossible to select a city for example. Using polygons from GeoJSON however, makes it possible to click on the respective polygon to use it as a filter in data analysis.

7.1.1 Datasets without a link to GeoJSON

Many datasets have natural attributes that can be linked to a GeoJSON-file, e.g., a city or country name. But this is not true for all datasets with geographical data. Some only contain coordinates as latitude and longitude attributes. To solve this problem, a geofencing algorithm can be used. Geofencing is the procedure of determining if a point is inside a polygon, which suits our use case perfectly. These algorithms usually rely on ray-casting and the rule saying that "if a point is inside a polygon, any ray cast from it will cross the polygons perimeter an odd number of times"[1]. This is applicable to the coordinates in a dataset and the polygons in a GeoJSON file. So far, we have used a utility script written in Python to do this ourselves, but it is not yet built into Slicer and remains as future work.

7.2 Incremental computation of data tiles

In the current implementation, data tiles are computed either from scratch if the size of the filtered dataset is assumed to be less than 50% the total size. Otherwise, the filtering is reversed and the tiles are computed by first creating tiles corresponding to opposite of the actual selection done client side which is then subtracted from the cached tiles corresponding to the unfiltered dataset. This ensures that the database never has to calculate aggregates on more than 50% of the dataset each time tiles are requested.

Crossfilter's observation that explorative selections often are incremental (subsets of subsets), could be translated to Slicer's task of creating data tiles. Falcon as well as this work has been mainly concerned with how to compute data tiles from a data set as quickly as possible. But it makes sense to ask the question how data tiles can be computed from existing data tiles as well. An incremental approach could be taken by caching tiles returned to clients. Then, when a client requests new tiles they can be computed as deltas from the most similar cached tiles. Thus reducing the portion of the dataset needed to calculate aggregates on could be reduced further than 50%. This would potentially lead to significant improvements in latency when swapping between which charts are filtered (which is what triggers a recalculation of tiles) since the aggregate functions are the most time consuming work done by the database and is tied directly to how many rows are read.

This caching approach would require some way to calculate the similarity between the filters active in the current request and the filters active in the cached tiles. A limit would also had to be set as to how many tiles can be cached. Because caching all possible tiles would end up creating essentially an in-memory data cube. Limiting how many tiles are to be cached further calls for a strategy to select which ones to cache. These questions stand as topics for future research and would benefit Falcon as well as this work. Even if Falcon already has very fast tile computation using a GPU-accelerated database, this

1. The geofencing algorithm is clearly explained here:
https://www.baeldung.com/cs/geofencing-point-inside-polygon

approach would be fully compatible with their work.

7.3 SQL performance

It is common to optimize query times by making one large query instead of many small ones[2]. In Slicer, one query is done for each data tile. We attempted to avoid this by combining all the queries into one and grouping on all the chart-dimensions. The result of that query was tuples on this form: (tile_id, x, y, value). The list of tuples was then iterated and the data tiles were updated according to id, x and y positions in the tile and the value. This approach was slower than running the queries separately, so it was abandoned.

Observing that all the queries operate on the same rows, another approach was also tested. We wanted to first query for the selected rows and then run the tile queries as sub-queries. The idea is the tile queries then only do aggregation and avoids redoing the same selecting for every tile. Databases have the concepts of *Views* and *Materialized views*. A view is functionality that makes it possible to store a query so that later queries can be on that view instead of directly on the tables. This is convenient for the developer because it reduces the amount of duplicated SQL code in the queries. It does not however provide any performance benefits (usually). Materialized views on the other hand are essentially new tables based on a view[3]. Creating these materialized views however comes with the cost of creating a new table, and we found that the overhead outweighed the speedup significantly. We found that the filtering takes a too small portion of the query times, and that the aggregation is the bottleneck.

7.4 Client-side tile engines

The only way to completely avoid network latency on some user interaction in Slicer, would be to generate the data tiles in the client. Doing so means sending the dataset to the client. For smaller datasets this might be sensible if the server has scarce resources or the application is known to be used by clients with varying network conditions.

2. See discussion at: `https://dba.stackexchange.com/questions/76973/what-is-faster-one-big-query-or-many-small-queries`
3. This is a simplification, and materialized views are usually used for different goals than sub-queries. More in-depth explanation can be seen at: `https://www.complexsql.com/materialized-view/`

Falcon has an implementation where the tiles are created on the client. The way they do it is by iterating the entire dataset and updating the corresponding values in the tiles along the way. This works fairly well up to 1 million records in our testing (beyond that, it can take >0.75 sec to generate the tiles). Slicer's front-end code is written to be agnostic as to where the tiles comes from, so changing out the tile engine from the server-side is trivial. Two ways of creating tiles client-side have been considered so far.

The first idea for a client-side engine revolves around the fact that DuckDB has been successfully compiled to WebAssembly[4]. If the backend-code responsible for creating the data tiles is rewritten in a language that can be compiled to WebAssembly, it is thinkable that the back-end and front-end engines can share most, if not all, of their code. There exists runtimes such as wasmtime[5], which could make it possible to run the same binary on the client (a web browser) and the server.

The second idea is based on one of the first prototypes for Slicer. The first prototype was purely client-side and not tile based. It uses WebGL[37] to filter and aggregate the data as well as rendering visualizations. To be able to use WebGL, the datasets were encoded into textures and sent to the clients. This implementation was able to filter, aggregate, and render 5 million data points in 10ms on the same laptop used in our experiments. Although this was with only one bar chart view and two possible filters, we thought the performance was very promising. The prototype was abandoned however, because it was considered more important to be able to use datasets that are too large to send to the client. Therefore a server-side solution was prioritized. After creating the tile-based version of Slicer, we now realize that the GPU-approach might be used to create the data tiles. And since WebGL is a subset of OpenGL, a tile engine is made this way can be run both in the clients browser and on the server.

7.5 Datasets too large to keep in-memory

Slicer currently uses an in-memory DuckDB instance to make data tiles. Many datasets will be too large to fit into memory - depending on the capacity of the server and the size of the dataset. Relying on main memory, was made as a trade-off to avoid relying on video memory, which is considerably more

4. See announcement at: `https://github.com/duckdb/duckdb/pull/1424`
5. Wasmtime website: `https://wasmtime.dev/`

expensive[6]. Still, for many applications it does not make sense to host a Slicer application using main memory. If the application is rarely used it may be seen as a waste of resources or the server might simply not have the capacity to keep the entire database in memory.

For instances where this is true, DuckDB has a convenient mode of operation available: DuckDB can be run with the data itself residing on disk while keeping indexes in memory. This keeps the performance of the filtering and grouping the same, while slowing down the aggregate functions somewhat (depending on the size of the dataset and read-speed of the disk). This is a both a feasible and agreeable solution as is, and will be exceedingly appealing if the incremental computation discussed in Section **??** yields results. That would further minimize how much needs to be read from disk, mitigating the performance cost of keeping the database on disk. It is presumable that such a mode of operation will be made default for Slicer in the future. Keeping the database on disk frees up memory which can be used to cache more data tiles. It is imaginable that this could lead to performance increases that transcends the benefit of having the database in-memory. This however, is left as a future research question.

Using another SQL-database altogether is also possible if none of the DuckDB approaches are convincing or if it makes sense to use a shared database for Slicer and other applications. In situations where the dataset is already stored in an SQL-database it can be redundant use of resources for Slicer to create its own database. The SQL used to compute the data tiles consists solely of basic SQL operations, which makes switching out the database engine as trivial as replacing the connection string. This means that Slicer can be configured to use sampling or GPU accelerated databases as BlinkDB or OmisciDB (which Falcon uses). DuckDB will be kept as default for its combination of performance and convenience from being in-process.

6. A curent generation GPU with 24GB vRAM costs around $2000, while 32GB main memory cost from around $160 and up. See: `https://www.nvidia.com/nb-no/geforce/graphics-cards/30-series/rtx-3090/` and `https://www.prisjakt.no/c/ddr4-minner?rg_95336=32-32&sort=price`. (Both urls visited August 2021)

8 Conclusion

8.1 Concluding remarks

In this work we have presented the design and implementation of Slicer, a framework for building CMV-style interactive applications for data exploration. The design builds upon earlier works in the field to provide similar performance with more features and simpler setup. Through exploring trade-offs and optimizations Slicer has further validated the data tile approach to data exploration: More types of visualizations can be based on them, they can support more aggregate functions than *Count*, and they do not need to be limited to *brushing* but can be used with *single-select* as well. Slicer also opens the approach to more datasets. Many datasets are to big to handle client-side, but not big enough to justify investing in a GPU-accelerated server[1].

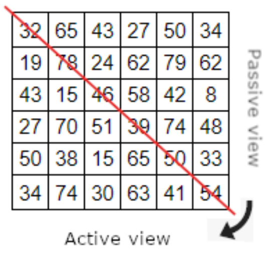
Unsatisfied with existing solutions, we set out to find and validate a method for creating better CMV-applications for data exploration. Our findings show that Slicer can offer satisfactory performance and functionality for our use cases. Yet we still have ideas for improvement and further research. These are outlined in Section 8.2.

1. Falcon's mechanism to quickly compute data tiles for larger datasets relies on OmniSciDB.

8.2 Future work

8.2.1 Exploit the symmetry of Slicer's data tiles

Slicer data tiles are diagonally symmetric; if flipped along the axis show in Figure 8.1 a tile can switch between which view is active and which is passive. The obvious implication of this when switching views without filtering, the tile between the old and new active view does not have to be computed. It can simply be flipped. In addition to this observation, we wish to investigate if further performance increases can come from the symmetric property.



32	65	43	27	50	34
19	78	24	62	79	62
43	15	46	58	42	8
27	70	51	39	74	48
50	38	15	65	50	33
34	74	30	63	41	54

Figure 8.1: Illustration of the symmetric nature of data tiles in Slicer.

8.2.2 Further utilization of bin arrays

Bin arrays are elementary in Slicer. It is what the visualizations are based on and what the data tiles aim to compute. We have capitalized on the observation that they can be used in cooperation with the data tiles to update themselves based on current and precious filters.

We wish to flip the idea that data tiles compute the bin arrays, and investigate if bin arrays can help compute data tiles. This idea is based on the fact that all bin arrays are known at any time in a Slicer-application - that is how the visualizations are made. In a data tile, the bins for the active and passive views are the same as its column- and row-sums respectively. Row- and column-sums can be used to create matrices that satisfy the sums. The problem is that there are multiple possible matrices for any set of sums[2]. What becomes a topic for future research is to examine if the sums can be combined with other factors making it possible to find the correct matrix. If these factors can be computed fast, it might lead to significant improvements in data tile computation.

2. This problem can be seen discussed here: `https://math.stackexchange.com/questions/1969542/find-a-matrix-with-given-row-and-column-sums`

8.2.3 Chart improvements

The bar charts in Slicer are currently missing labels on the x- and y-axes. This not because the labels are unavailable, it has just not been implemented. A proprietary HTML-canvas[3] renderer was created since the most existing ones were based on D3[4] and thus were SVG-based. SVG-based renderers creates DOM-elements for each part of the visualizations. DOM-elements have overhead on creation and update, and clutters the DOM-tree degrading the performance of the entire website[5]. We will therefore keep our canvas renderer and add the axis labels using D3, in the same way we have used its *brush* implementation.

8.2.4 Client-side caching of tiles

In the current implementation of Slicer, each time a new active view is set, new tiles are requested from the server. This happens regardless of new filters being set or not. Slicer's data tiles range from bytes to kilobytes in size, while browser tabs have one to several gigabytes memory available[6]. A user moving the cursor around in the application will trigger new active views, but it does not necessarily mean that any filtering has been done. This makes recomputing the data tiles every time a user moves the cursor between two views without changing any filters redundant. Instead, all tiles could be cached as long as no new filters are applied - and when they are, the cache can be cleared out.

An example of a situation where this will be beneficial is when a user makes a selection in a chart and then goes on to zoom-in or pan in a map view (without making any selections in it). Then after adjusting the map view, the user goes on to make further selections in the initial chart. Currently, moving the cursor to the map view triggers a re-computation of tiles, and then another one is triggered when the cursor is moved back to the chart. This last re-computation can be avoided by caching the tiles until the filters are changed.

3. See: `https://www.w3schools.com/html/html5_canvas.asp`
4. See: `https://d3js.org/`
5. See discussion at: `https://stackoverflow.com/questions/5882716/html5-canvas-vs-svg-vs-div`
6. See discussion about RAM available in tabs here: `https://stackoverflow.com/questions/29620041/is-there-any-memory-limit-for-google-chrome-browser`

8.2.5 Speeding up tile generation

We are satisfied with the interactivity of Slicer applications when interacting with the data tiles. And for datasets up to the tested sizes we are content with the delay taken to compute new data tiles when changing active view. We wish to test the system with larger data sets and investigate possible avenues for improvement on data tile computation. Primarily, we intend to investigate the iterative approach as discussed in Section 7.2.

Concurrent database queries is another other avenue of interest. Since the database is loaded as read-only, concurrent and possibly parallel querying of it is possible. Implementing this would reduce the time to create the requested data tiles and improve how many concurrent clients the server can handle.

Inspecting Figure 6.2, we see that tile computation does not scale linearly for smaller dataset sizes. Investigating the reason for this, might open up for improvements. If the scaling cannot be improved, optimal database sizes for datasets might be investigated instead. Datasets can be divided in subsets, each with their own DuckDB instance, which can either work parallel, or sequentially to improve on performance.

References

[1] H. Koh and C. Magee. “A functional approach for studying technological progress: Application to information technology.” In: *Technological Forecasting and Social Change* 73 (2006), pp. 1061–1083.

[2] AI Impacts. *Trends in the cost of computing*. URL: https://aiimpacts.org/trends-in-the-cost-of-computing/ (visited on 01/30/2021).

[3] John C. McCallum. *Disk Drive Prices 1955+*. URL: https://jcmit.net/diskprice.htm (visited on 01/30/2021).

[4] Andy Klein. *Hard Drive Cost Per Gigabyte*. URL: https://www.backblaze.com/blog/hard-drive-cost-per-gigabyte/ (visited on 01/30/2021).

[5] Henry F. Korth Abraham Silberschatz and S. Sudarshan. *Database System Concepts 7th ed.* New York: McGraw-Hill, 2019.

[6] Microsoft. *PowerBI*. URL: https://powerbi.microsoft.com/ (visited on 01/29/2021).

[7] Google Cloud. *BigQuery*. URL: https://cloud.google.com/bigquery/ (visited on 01/29/2021).

[8] Tableau. *Tableau*. URL: https://www.tableau.com/ (visited on 01/29/2021).

[9] J. C. Roberts. “State of the Art: Coordinated Multiple Views in Exploratory Visualization.” In: *Fifth International Conference on Coordinated and Multiple Views in Exploratory Visualization (CMV 2007)*. 2007, pp. 61–71. DOI: 10.1109/CMV.2007.20.

[10] Emanuel Zgraggen et al. “How progressive visualizations affect exploratory analysis.” In: *IEEE transactions on visualization and computer graphics* 23.8 (2016), pp. 1977–1987.

[11] Zhicheng Liu and Jeffrey Heer. “The Effects of Interactive Latency on Exploratory Visual Analysis.” In: *IEEE Transactions on Visualization and Computer Graphics* 20.12 (2014), pp. 2122–2131. DOI: 10.1109/TVCG.2014.2346452.

[12] Albert Ng et al. “Designing for Low-Latency Direct-Touch Input.” In: *Proceedings of the 25th Annual ACM Symposium on User Interface Software and Technology*. UIST ’12. Cambridge, Massachusetts, USA: Association for Computing Machinery, 2012, 453–464. ISBN: 9781450315807. DOI: 10.1145/2380116.2380174. URL: https://doi.org/10.1145/2380116.2380174.

[13] Zhicheng Liu, Biye Jiang, and Jeffrey Heer. “imMens: Real-time Visual Querying of Big Data.” In: *Computer Graphics Forum* 32.3pt4 (2013), pp. 421–430. DOI: https://doi.org/10.1111/cgf.12129. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1111/cgf.12129.

[14] Dominik Moritz, Bill Howe, and Jeffrey Heer. “Falcon: Balancing Interactive Latency and Resolution Sensitivity for Scalable Linked Visualizations.” In: *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems - CHI '19*. ACM Press, 2019. DOI: 10.1145/3290605.

[15] Square. *Crossfilter*. URL: https://square.github.io/crossfilter/ (visited on 01/29/2021).

[16] Highcharts. *Highcharts*. URL: https://www.highcharts.com/ (visited on 01/24/2021).

[17] Peter J. Denning et al. “Computing as a discipline.” In: *Computer* 22.2 (1989), pp. 63–70.

[18] Alan R Hevner et al. “Design science in information systems research.” In: *MIS quarterly* (2004), pp. 75–105.

[19] Sameer Agarwal et al. “BlinkDB: Queries with Bounded Errors and Bounded Response Times on Very Large Data.” In: *Proceedings of the 8th ACM European Conference on Computer Systems*. EuroSys ’13. Prague, Czech Republic: Association for Computing Machinery, 2013, 29–42. ISBN: 9781450319942. DOI: 10.1145/2465351.2465355. URL: https://doi.org/10.1145/2465351.2465355.

[20] Sameer Agarwal et al. *BinkDB*. URL: http://blinkdb.org/ (visited on 01/23/2021).

[21] H. Butler et al. *The GeoJSON Format*. RFC 7946. Aug. 2016. DOI: 10.17487/RFC7946. URL: https://rfc-editor.org/rfc/rfc7946.txt.

[22] I. Fette et al. *The WebSocket Protocol*. URL: https://datatracker.ietf.org/doc/html/rfc6455 (visited on 05/24/2021).

[23] Saurabh. *WebSocket vs HTTP Calls - Performance Study*. URL: https://browsee.io/blog/websocket-vs-http-calls-performance-study/ (visited on 05/24/2021).

[24] Python Software Foundation. *Python*. URL: https://www.python.org/ (visited on 06/15/2021).

[25] NumPy. *NumPy*. URL: https://numpy.org/ (visited on 06/15/2021).

[26] Pandas. *Pandas*. URL: https://pandas.pydata.org/ (visited on 06/15/2021).

[27] Ami Marowka. “Python accelerators for high-performance computing.” In: *The Journal of Supercomputing* 74.4 (2018), pp. 1449–1460.

[28] Sebastián Ramírez (alias: tiangolo). *FastAPI*. URL: https://fastapi.tiangolo.com/ (visited on 06/15/2021).

[29] Microsoft. *TypeScript*. URL: https://www.typescriptlang.org/ (visited on 06/15/2021).

[30] R.C. Martin. *Clean Code: A Handbook of Agile Software Craftsmanship*. Robert C. Martin Series. Pearson Education, 2008. ISBN: 9780136083252. URL: https://books.google.no/books?id=_i6bDeoCQzsC.

[31] E. Gamma et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Pearson Education, 1994. ISBN: 9780321700698. URL: https://books.google.no/books?id=6oHuKQe3TjQC.

[32] Web Hypertext Application Technology Working Group. *HTML Canvas Specification*. URL: https://html.spec.whatwg.org/multipage/canvas.html#the-canvas-element (visited on 04/24/2021).

[33] Ecma International. *Promise Objects Spesification*. URL: https://tc39.es/ecma262/multipage/control-abstraction-objects.html#sec-promise-objects (visited on 06/02/2021).

[34] Henrik Nielsen et al. *Hypertext Transfer Protocol – HTTP/1.1*. RFC 2616. June 1999. DOI: 10.17487/RFC2616. URL: https://rfc-editor.org/rfc/rfc2616.txt.

[35] Dr. Mark Raasveldt and Dr. Hannes Mühleisen et al. *DuckDB*. URL: https://duckdb.org/ (visited on 03/12/2021).

[36] Lauro Lins, James T Klosowski, and Carlos Scheidegger. "Nanocubes for real-time exploration of spatiotemporal datasets." In: *IEEE Transactions on Visualization and Computer Graphics* 19.12 (2013), pp. 2456–2465.

[37] Khronos Group. *WebGL Specification*. URL: https://www.khronos.org/registry/webgl/specs/latest/1.0/ (visited on 04/24/2021).

A

Code for generating SQL queries

Code Listing A.1: This code generates the SQL needed to query for data to create a data tile from an active view to a passive view with filters. This code is vulnerable to SQL-injection and needs to be rewritten to be parameterized before being used in production.

```python
def get_tile_sql(self, from_view, to_view, filters):
    x_bin = f"{from_view.name}_bin"
    y_bin = f"{to_view.name}_bin"
    if to_view.reduce == "sum":
        reduction = f"SUM({to_view.y_dimension})"
    elif to_view.reduce == "avg":
        reduction = f"AVG({to_view.y_dimension})"
    else:
        reduction = f"COUNT(*)"
    filter_sql = self.filters_to_sql(filters, from_view.name, to_view.name)
    return f"SELECT {x_bin} AS x, {y_bin} AS y, {reduction} AS v " \
           f"FROM Data {filter_sql} " \
           f"GROUP BY x, y"

def filters_to_sql(filters, from_view, to_view):
    if not filters:
        return ""
    sql_filters = []
    for view_name, f in filters.items():
        if view_name == from_view or view_name == to_view:
            continue
        if f['type'] == FilterType.Range:
            sql_filters.append(f"({view_name}_bin BETWEEN"
                               f" {f['range'][0]} AND {f['range'][1]})")
        elif f['type'] == FilterType.Categorical:
            cats = [str(c) for c in f['categories']]
            sql_filters.append(f"({view_name}_bin IN ({','.join(cats)}))")
    if sql_filters:
        return "WHERE " + " AND ".join(sql_filters)
    else:
        return ""
```

B

Code for generating data tiles (with timing)

Code Listing B.1: DuckDbDataProvider.get_tile() method (with timing).

```
def get_tile(self, from_view, to_view, filters=None):
    t0 = time.time()
    f = self.config.views[from_view]
    t = self.config.views[to_view]
    cur = self.conn.cursor()
    sql = self.get_tile_sql(f, t, filters)
    cur.execute(sql)
    res = cur.fetchnumpy()
    cur.close()
    res = self.result_to_nparray(res, f, t)
    print(f"Creating tile for {to_view} took:", time.time()-t0)
    return res
```

/C Extract from slicer-frontend/src/Mediator.ts

Code Listing C.1: Extract from slicer-frontend/src/Mediator.ts

```
export enum MediatorSubject {
    SetActive,
    NewActive,
    InitialDataReady,
    Filter
}

export type MediatorMsg = {
    subject: MediatorSubject,
    content?: string | Filter
}

export function subscribe(subject: MediatorSubject, cb: Callback) {
    Mediator.instance.subscribe(subject, cb);
}

export function send(msg: MediatorMsg) {
    Mediator.instance.send(msg);
}
```

D Extract from slicer-frontend/src/Types.ts

Code Listing D.1: Extract from slicer-frontend/src/Types.ts

```
export interface AppConfig {
    app: string;
    views: Record<string, ViewConfig>;
    tiles: TileConfig[];
}

export interface ChartConfig {
    name: string;
    type: "barchart";
    dimension: string;
    nbuckets: number;
    lookup: string[];
}

export interface LayerConfig {
    name: string;
    type: "map_layer";
    dimension: string;
    nbuckets: number;
    lookup: {[key: string]: number};
    key: string;
    geojson: string;
}

export type ViewConfig = ChartConfig | LayerConfig;

export interface TileConfig {
    name: string;
    x: string;
    y: string;
    w: number;
    h: number;
}
```

E

Extract 2 from slicer-frontend/src/Types.ts

Code Listing E.1: Extract 2 from slicer-frontend/src/Types.ts

```
export enum FilterType {
    Range = 0,
    Categorical = 1
}

export interface RangeFilter {
    type: FilterType.Range,
    view: string,
    range: Range
}

export interface CategoryFilter {
    type: FilterType.Categorical,
    view: string,
    categories: number[]
}

export type Filter = RangeFilter | CategoryFilter;
```

F

Code for estimating row count based on filter

Code Listing F.1: Code for estimating row count based on filters

```python
def estimate_selected_row_count(self, filters):
    lowest_count = math.inf
    for view_name, f in filters.items():
        count = 0
        hist = self.histograms[self.view_lookup[view_name]]
        view = self.views[view_name]
        if f['type'] == "range":
            for i in range(f['range'][0], f['range'][1]):
                count += hist[i]
        elif f['type'] == "categorical":
            for cat in f['categories']:
                count += hist[view.order_lookup[cat]]
        lowest_count = min(count, lowest_count)
    return lowest_count
```