UiT
THE ARCTIC UNIVERSITY OF NORWAY

Faculty of Science and Technology
Department of Computer Science

Metadata state and history service for datasets

Enable extracting, storing and access to metadata about a dataset over time.

—

Roberth Hansen

INF-3990 Master's Thesis in Computer Science - May 2018

This thesis document was typeset using the *UiT Thesis LaTeX Template*.
© 2018 – http://github.com/egraff/uit-thesis

To Maria.

Thank you very much.

“When I’m working on a problem, I never think about beauty.
I think only how to solve the problem.
But when I have finished, if the solution is not beautiful,
I know it is wrong.”
–R. Buckminster Fuller

“The most important property of a program
is whether it accomplishes the intention of its user.”
–C.A.R Hoare

Abstract

Distributed Arctic Observatory (DAO) aims to automate, streamline and improve the collection, storage and analysis of images, video and weather measurements taken on the arctic tundra. Automating the process means that there are no human users that needs to be involved in the process. This leads to a loss of monitoring capabilities of the process. There are insufficient tools that allow the human user to monitor the process and analyze the collected volume of data.

This dissertation presents a prototype of a system to aid researchers in monitoring and analyzing metadata about a dataset. The approach is a system that collects metadata over time, stores it in-memory and visualizes the metadata to a human user.

The architecture comprises three abstractions Dataset, Instrument and Visualization. The Dataset contains metadata. The Instrument extracts the metadata. The Instrument supplies metadata to the Visualization abstraction.

The design comprises a Dataset, Metadata extractor, Dataset server, Web server and Visualization. The Dataset is a file system. The Metadata extractor collects metadata from the dataset. The Dataset server stores the collected metadata. The Web server requests metadata from the dataset server and supplies it to a web browser. The Visualization uses the metadata to create visualizations.

The Metadata extractor is a prototype written in Python and is executed manually as a process. The Dataset server utilizes Redis as an in-memory database and Redis is executed manually as a process. Redis supports a selection of data structures, this enables a logical mapping of metadata. The Web server is implemented using the Django web framework and is served by Gunicorn and Nginx. The Visualization is implemented in JavaScript, mainly utilizing Google Charts to create the visualizations.

A set of experiments was conducted to document performance metrics for the prototype. The results show that we can serve about 2500 web pages to 10

concurrent connections with a latency below 5 ms. The results show that we can store 100 million key-value pairs in 9 GB of memory. Our calculations indicates that it will take over 690 years to reach 9 GB of memory footprint with the current structure of metadata.

This dissertation designs, implements and evaluates an artifact prototype that allow researcher to monitor and analyze metadata about a dataset over time. We contribute an architecture and design that enables and supports the creation of visualizations of organized and processed metadata. The artifact validates using in-memory storage to store the historic metadata.

// Acknowledgements

I would like to thank my main advisor Professor Otto Anshus, and co-advisor Associate professor John Markus Bjørndalen for you advice, ideas and feedback. I want to especially thank Otto for our hours long discussions about defining the architecture and design of the system, and of course naming things.

I want to express my gratitude to my fellow students, especially Simon who pulled me through my first three years. And Nina who's been a valuable discussion partner.

I would like to thank my dad for always encouraging me and a special thanks to my mom who has been taking care of my son Tobias when I've been writing this thesis.

Tobias, you are the reason I do this.

Maria, you have been there for me for every step of this five year journey. Thank you for listening to me,

But I

I love it when you read to me

And you

You can read me anything

Contents

Abstract iii

Acknowledgements v

List of Figures xi

List of Tables xiii
Listings xiii

List of Listings xv

List of Abbreviations xvii

1 Introduction 1
1.1 Challenges 2
1.2 Main contributions 2
1.3 Outline 3

2 Related work 5

3 Idea 7

4 Architecture 9
4.1 Human user abstraction 9
4.2 Visualization abstraction 11
4.2.1 Interact with human user 11
4.2.2 Request metadata 11
4.2.3 Transform metadata 11
4.2.4 Visualize information for user 12
4.3 Instrument 12
4.3.1 Locate metadata dataset 12
4.3.2 Collect and return metadata 12
4.3.3 Metadata dataset 12
4.3.4 Extract metadata 13

4.4 Dataset 13

5 Design 15

5.1 Visualization 15

5.1.1 Visualization application 17

5.1.2 Web browser client 17

5.2 Web server 17

5.3 Dataset service 18

5.4 Dataset server 19

5.5 Metadata extractor 19

6 Implementation 21

6.1 Visualization 21

6.1.1 Technologies 21

6.1.2 Information types 23

6.1.3 Directory names 25

6.2 Web server 25

6.2.1 Commands 26

6.2.2 Response 26

6.3 Dataset server 27

6.3.1 Commands 27

6.3.2 Redis data structures 28

6.3.3 Redis pipelines 29

6.4 Metadata extractor 29

6.4.1 File system iteration 30

6.4.2 Metadata extraction 30

6.4.3 Unique hash ID 31

6.5 Technologies 31

6.6 File creation 31

7 Experiments 35

7.1 Benchmarking tools 36

7.1.1 wrk 36

7.1.2 psutil 38

7.1.3 Redis benchmark 38

7.2 Experiment dataset 38

7.3 Web browser client 39

7.4 Web browser client - memory footprint 40

7.4.1 Methodology 40

7.4.2 Metrics 40

7.5 Web browser client - network usage 40

7.5.1 Methodology 40

7.5.2 Metrics 40

7.6 Web server 40

7.7 Web server - requests per second 42
7.7.1 Methodology 42
7.7.2 Metrics 42
7.8 Web server - latency 42
7.8.1 Methodology 42
7.8.2 Metrics 42
7.9 Web server - CPU usage 43
7.9.1 Methodology 43
7.9.2 Metrics 44
7.10 Dataset server - keys 44
7.10.1 Methodology 44
7.10.2 Metrics 45
7.11 Dataset server - memory utilization 45
7.11.1 Methodology 45
7.11.2 Metrics 46
7.12 Dataset server - CPU utilization 46
7.12.1 Methodology 46
7.12.2 Metrics 46
7.13 Dataset server - requests per second 47
7.13.1 Methodology 47
7.13.2 Metrics 47
7.14 Metadata extractor - execution time 47
7.14.1 Methodology 47
7.14.2 Metrics 48
7.15 Metadata extractor - resource usage 48
7.15.1 Methodology 48
7.15.2 Metrics 49
7.16 System - Resource usage 49
7.16.1 Methodology 49
7.16.2 Metrics 50
7.17 os.walk 50
7.17.1 Methodology 50
7.17.2 Metrics 50
7.18 Reported disk usage 51
7.18.1 Methodology 51
7.18.2 Metrics 51

8 Results 53
8.1 Web browser client - memory footprint 53
8.2 Web browser client - network usage 54
8.3 Web server - web pages 55
8.4 Web server - Commands 56
8.5 Web server - CPU utilization 58
8.6 Dataset server - keys 60

8.7 Dataset server - memory utilization 61
8.8 Dataset server - CPU utilization 62
8.9 Dataset server - requests per second 63
8.10 Metadata extractor - execution time 63
8.11 Metadata extractor - resource usage 65
8.12 System - CPU utilization 66
8.13 System - memory footprint 68
8.14 os.walk 69
8.15 Reported disk usage 70

9 Discussion 73
9.1 Thesis 73
9.2 Optimizing Redis 74
9.3 Scale 75
9.4 The amount of keys 76
9.4.1 One year of measurements 76
9.4.2 Dataset growth 77
9.5 Prototype bottleneck 78
9.6 React, abstractions and Google charts 78
9.7 Extract metadata on dataset change 79
9.8 Metadata extractor resource usage 79

10 Contributions 81

11 Summary and Conclusion 83

12 Future work 85

A The Road Towards the Artifact 87
A.1 Approach 1 88
A.2 Approach 2 89
A.3 Approach 3 91
A.4 Conclusion 92

B Redis key size 93

C Redis mass-insertion 95

Bibliography 97

List of Figures

3.1 The system Idea 8

4.1 System architecture. 10

5.1 System design 16

6.1 System implementation. Each blue square is a process. . . . 22
6.2 Redis hash 28
6.3 Redis sorted set 29

7.1 wrk sample output 37

8.1 Web server requests per second and latency for delivering web pages 56
8.2 Web server requests per second and latency for responding to commands 57
8.3 Gunicorn CPU utilization 59
8.4 nginx CPU utilization 59
8.5 Dataset sever memory utilization 61
8.6 Dataset server CPU utilization 62
8.7 Metadata extractor execution time for Small, Medium and Big datasets 65
8.8 Metadata extractor CPU utilization and memory utilization . 66
8.9 System - CPU utilization in an idle state 67
8.10 System - CPU utilization under load 68
8.11 System memory utilization 69
8.12 os.walk execution time in both Python 2.6 and Python 3.6 . 70

A.1 Version 1 square placement 89
A.2 Version 2 static placement 90
A.3 Version 2 dynamic placement 90
A.4 Circular placement 91

List of Tables

6.1 Computer specifications 32

7.1 Dataset differences 39

8.1 Web client memory footprint 53
8.2 Web client network usage 54
8.3 Biggest data types 60
8.4 Aggregate data type information 60
8.5 Total keyspace information 61
8.6 Requests per second from Redis benchmark 64
8.7 Reported disk usage 71

9.1 Calculated memory utilization 76
9.2 Theoretical memory footprint of Big dataset 77

A.1 Directory size quota 88

List of Listings

6.1 JSON response from total number of files command. 23
6.2 Example JSON response from metadata command. 24
6.3 Example JSON response from history command. 24
6.4 os.walk() function 30
6.5 hash function 31
6.6 File creation 32

7.1 wrk command 37
7.2 Redis benchmark command 38
7.3 psutil capture of CPU usage 43
7.4 Redis bigkeys command 44
7.5 Mass insertion command 45
7.6 Redis benchmark command 47
7.7 Metadata extractor execution time 47
7.8 Metadata extractor resources usage 48
7.9 os.walk experiment 50
7.10 Disk usage measurement 52

B.1 https://gist.github.com/epicserve/5699837 93

C.1 https://github.com/TimSimmons/redis-mass-insertion . . . 95

List of Abbreviations

b Byte

CLI Command-Line Interface

COAT Climate-ecological Observatory for Arctic Tundra

DBMS Database Management System

GB Gigabyte

HTTP Hypertext Transfer Protocol

IoT Internet of Things

JSON JavaScript Object Notation

kb Kilobyte

mb Megabyte

ms Milliseconds

PID Process identifier

REST Representational State Transfer

s Seconds

TCP Transmission Control Protocol

URL Uniform Resource Locator

Introduction

Climate-ecological Observatory for Arctic Tundra (COAT) is a long-term, ecosystem-based and adaptive observation system. It aims to unravel how climate change impacts arctic tundra ecosystems and to enable prudent science-based management.

Distributed Arctic Observatory (DAO) is a project that aims to automate, streamline and improve the collection, storage and analysis of images, videos and weather measurements. The project is based around using custom IoT (Internet of Things) hardware and software that can automate the collection process and reporting. The stored data is classified using machine learning.

The automation of collection and storage of data means that no researcher will have direct control over the collection and storage process. Because the researcher is removed from the process, the researcher looses the ability to monitor the process. If one of the steps in the automation process fails, there are insufficient tools for a researcher to identify that a process has failed, and where it failed. There are no tools that enables monitoring the volume of data that is collected. Or that enables the analysis of information about the collected volume of data.

The data is assumed stored in a file system. A file system primarily enables a user to store, organize and access files. A file system has some metadata about itself and for each file stored in the file system. Some file systems gives the user the ability to get an overview of the disk usage, either by specifying what type

of files are using disk space [1] or by specifying which directories are using disk space [2].

If a researcher has access to metadata about the file system and a historic view of the metadata, this can enable a researcher to monitor and analyze the dataset. Our solution is to extract the metadata over time and visualize the extracted metadata.

1.1 Challenges

A user has no simple ways of getting an overview and detailed information about the state and history of a dataset. The user should be able to identify trends and changes in the dataset. This can help a user identify trends that are important for the understanding of the dataset. A user may want to look for items in the dataset that has a steeper increase of size or occurrences compared to the other items. If a user identifies a change that happens on a regular frequency, but has stopped now. Then the user can analyze it and determine why it stopped. This can indicate that something is wrong. We want to enable the user to accomplish these tasks by visualizing information. "The goal of visualization is to aid our understanding of data by leveraging the human visual systems ability to see patterns, spot trends and identify outliers."[9]

The dataset is assumed stored on commodity hardware that is used primarily for storage. This leaves unused computer resources that we can utilize to create a system that helps the user accomplish some of these tasks. The system will create an overview of a dataset, detailed information about the dataset and a historic representation of the dataset over time.

To represent a dataset over time, we create a system that extracts information from the dataset at periodic intervals over a period of time. The extraction process gathers information about the dataset to create an overview. And collects information about each item in the dataset to create detailed information. The extracted information is stored in an in-memory database. We can use the stored metadata to create a visualization.

1.2 Main contributions

This thesis makes the following contributions:

- An description of the approach and issues seen while progressing towards the described artifact prototype.
- An architecture and design of a system that uses in-memory storage to store information about a dataset over time.
- An architecture and design of a system that enables human users to interact with and visualize stored information.
- Implementation of the artifact prototype system
- An evaluation of the system identifying the CPU utilization, memory footprint and network activity between the parts of the system.
- Validating the idea of using in-memory storage to store dataset information over time.
- Thoughts on future work and further improvements to the current prototype.

1.3 Outline

The remainder of the thesis is structured as follows.

Chapter 2 presents Related work that covers large-scale data analytics, metadata management of large storage systems and implementing file systems in DBMS. The related work also covers visualization techniques.

Chapter 3 describes the basic Idea of the project. Detailing how the idea is split into four divisions of concern: Human User, View, Information and Dataset.

Chapter 4 presents the Architecture of the system. This includes every major functionality that the prototype offers.

Chapter 5 presents the Design of the prototype. The design specifies each system in the prototype.

Chapter 6 details the prototype Implementation. We go through each system and describes how it is implemented.

Chapter 7 describes the Experiments that we will perform. The experiments covers each system in the prototype.

Chapter 8 presents the Results of the experiments.

Chapter 9 presents a Discussion about the project and the prototype.

Chapter 10 details the Contributions of this paper.

Chapter 11 is a Summary and Conclusion of the paper.

Chapter 12 describes Future work. We discuss the paths forward for the prototype.

Appendix A detail the approach and issues seen while progressing towards the described prototype. Includes a conclusion of the approach.

Related work

Implementing Filesystems by Tree-aware DBMSs Implementing Filesystems by Tree-aware DBMSs [6] presents research to query data stored in a filesystem by using semi-structured database technology. The paper focuses on the ability to search/find and access stored data. The paper is based on the idea of mapping a filesystem hierarchy to XML.

Synchronous Metadata Management of Large Storage Systems The paper Synchronous Metadata Management of Large Storage Systems [7] compares three different approaches to store metadata. The three approaches are; disk based relational database systems, main memory relational database systems, in-memory key-value databases. They find that for the types of queries they used, the in-memory key-value outperformed relational databases. This is because of the extra features a relational database provide compared to key-value stores. For their approach they claim that metadata stored in a database system is valuable only if its kept in sync with the corresponding filesystem.

Disco: A computing Platform for Large-Scale Data Analytics Disco: A computing Platform for Large-Scale Data Analytics [8] presents a distributed computing platform for MapReduce computations on a filesystem. Disco implements a distributed filesystem specialized for the MapReduce use case. The distributed filesystem is tag-based. Instead of an hierarchical directory based organization, sets of data objects are tagged with names (tags).

A tour through the Visualization Zoo A tour through the Visualization Zoo [9] presents a collection of visualization techniques. These techniques include geographical maps, cartograms, node-link diagrams, tree layout, treemap, nested circle and matrix.

Issues and Benefits of Using 3D Interfaces: Visual and Verbal Tasks. The paper Issues and Benefits of Using 3D Interfaces: Visual and Verbal Tasks [10] presents that the brain uses different parts to process icon's in 2D and 3D space.

A visualization Model for Web Sitemaps The paper A visualization Model for Web Sitemaps [11] presents a visualization model that retrieves relational links from a website and visualizes its sitemap. The visualization uses an enclosure and connection approach for visualizing hierarchical information.

Visualization of Large Hierarchical Data by Circle Packing The paper Visualization of Large Hierarchical Data by Circle Packing [12] presents a visualization model that uses nested circles. The radius of a circle represent the size.

The Eyes Have It: A Task by Data Type Taxonomy for Information Visualizations The paper The Eyes Have It: A Task by Data Type Taxonomy for Information Visualizations [13] presents the visual information-seeking mantra: overview first, zoom and filter, then details on demand. The paper also describes seven tasks that are useful for information-seeking: overview, zoom, filter, details-on-demand, relate, history, extract.

DeepEye: An Automatic Big Data Visualization Framework The paper DeepEye: An Automatic Big Data Visualization Framework [4] presents system for automatically choosing a visualization after transforming the dataset. They use machine learning to determine which type of visualization technique is best suited for a particular dataset.

Towards the Understanding of Interaction in Information Visualization The paper Towards the Understanding of Interaction in Information Visualization [5] presents a review of visualizations and the interaction with visualization. They propose a taxonomy of eleven categories of visual interaction techniques, that can help future research. The categories are: filtering, selecting, abstract/elaborate, overview and explore, connect/relate, history, extraction of features, reconfigure, encode, participation/collaboration, gamification.

3 Idea

The idea is to create a system that gives a human user a method to visually view information about a dataset. The method includes giving the user a method to track dataset changes over time. We split the approach we will use into four divisions of concern as shown in fig. 3.1.

The **Human User** wants to understand a dataset. If the Human User understands the dataset, he can use the understanding to analyze the data. If the Human User has access to information about the dataset over a period of time, he can use the periodic information to look for trends and identify changes over time in the dataset.

The **View** uses Information to create visualizations. The visualizations include well known visualization techniques such as column charts, line charts and tables. The View allows the user to give input that can control how a visualization is shown by changing the zoom level, sorting the information or filtering out some information. The input also controls which Information the View visualizes.

Information is organized and processed metadata about a Dataset. The Information stores collected metadata in-memory. The metadata is shaped into Information before it is given to the View. Metadata is collected from the Dataset on specific intervals over a period of time. This gives the Information a history of metadata about a Dataset.

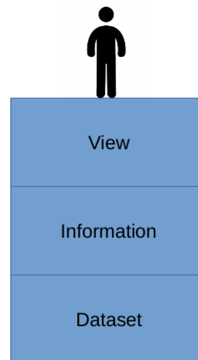


Figure 3.1: The system Idea

The **Dataset** is a volume of data that changes over time. The Dataset includes different types of elements. Each element has some metadata associated with itself.

4 Architecture

The system architecture is shown in fig. 4.1.

4.1 Human user abstraction

The system will have human users that interact with the visualization of information about a dataset. We assume that the human user can absorb information in a visual manner. The human user want to visualize information to better understand the dataset. If the human user understand the information he can gain insight into the dataset and use the information to analyze the dataset. The human user wants to see information about the dataset over a period of time. This will give the human user the ability to see trends in the dataset. If the human user can identify certain trends, he can identify if something doesn't fit within the trend. Viewing information over a period of time can also be used to monitor the dataset. If the human user knowns that a certain change happens on a set interval, then he can monitor the dataset to see if the change happens.

The human user wants to provide input to change the view of the information. The input can:

- change which information is used to create the visualization.

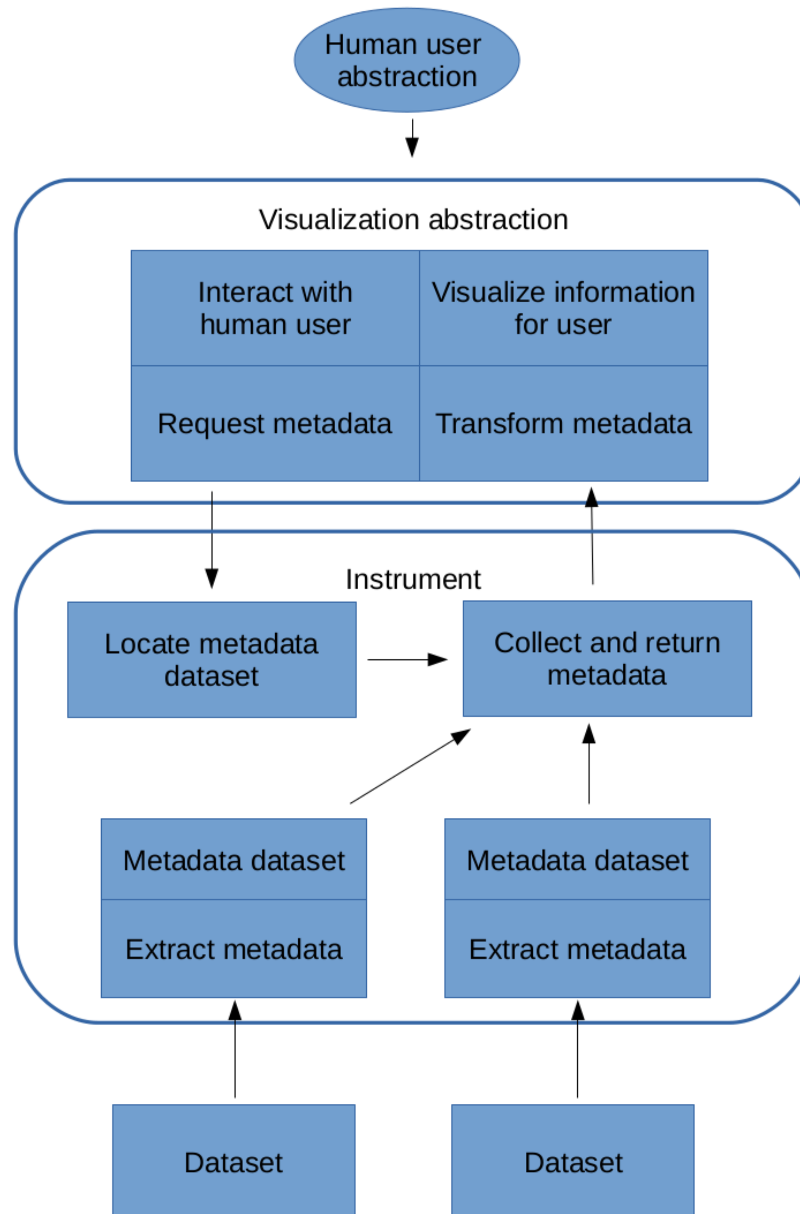


Figure 4.1: System architecture.

- add more information to an existing visualization.
- change the way the information is visualized. This includes filtering and sorting.

4.2 Visualization abstraction

4.2.1 Interact with human user

The visualization abstraction gives the human user options to give input to the visualization. The input allows the user to change how the information is visualized. If the visualization uses a specific visualization technique, the input can change the technique that is used. The input can change how numbers are represented if they are difficult for the human user to understand. The visualization can change the visualization based on input about which time period to visualize.

The input also allows the human user to give input about which information is visualized. The input can add more information to the current visualization or it can change the type of information that is visualized.

4.2.2 Request metadata

If the visualization abstraction receives input that require new information, the visualization abstraction must request the new information. The visualization abstraction requests new information from the Instrument. Each information request consist of the target dataset and the requested information. The requested information can include multiple metadata parts. The Instrument only stores metadata. The visualization abstraction sends requests for each metadata part including the target dataset to the Instrument.

4.2.3 Transform metadata

When each metadata part is returned from the Instrument, the visualization abstraction must combine them into the requested information. When the visualization abstraction combines the metadata into information, the metadata must be transformed into a format that the visualization can understand. The transformation includes converting strings to the appropriate data format such as integers and dates. The transformation also includes sorting the metadata elements.

4.2.4 Visualize information for user

The visualization abstraction uses the information that is combined and transformed from metadata, to create visualizations for the human user. The visualization abstraction stores the information that it uses to create visualizations in memory. Based on the information that is requested, the visualization abstraction creates different types of visualization. Based on the type of visualization that is used, the visualization abstraction creates different controls that can take input from the human user.

4.3 Instrument

4.3.1 Locate metadata dataset

The Instrument receives requests from the visualization abstraction. The request includes the requested metadata and the target metadata dataset. In principle there are multiple metadata datasets in the system. The Instrument collects metadata from the target metadata dataset.

4.3.2 Collect and return metadata

Each request for metadata is a new metadata dataset collection procedure. The collection procedure finds the requested metadata in the target metadata dataset. When the collection procedure is finished, the metadata is returned to the visualization abstraction.

4.3.3 Metadata dataset

The metadata dataset contains a history of metadata about a dataset and each element within the dataset. The metadata dataset stores metadata for set intervals over a period of time. The metadata dataset is stored in-memory. The metadata dataset include **aggregate metadata** about the dataset and **individual metadata** about each element.

Aggregate metadata of interest includes:

- Total size of a dataset
- Total number of items in a dataset

- Element types
- Total number of each element type

Individual metadata of interest include:

- Name
- Date modified
- Size
- List of items within an element
- Number of items within an element

4.3.4 Extract metadata

The metadata that is stored in the metadata dataset is extracted from the dataset. The extraction process collects aggregate metadata about the dataset and the individual metadata for each element. The extraction process also collects information about the structure of the dataset.

4.4 Dataset

The dataset is a data volume that contain different elements. Each element has some metadata associated with itself. The dataset is structured in way that some elements creates the structure and some elements are contained within the structure. The elements that create the structure of the dataset, know who they are connected to. The elements that are contained within the structure, only know which structural element they are connected to. The dataset changes over time, with elements being removed and added to the dataset.

5

Design

The design of the system is shown in fig. 5.1.

5.1 Visualization

The visualization are the visual tool that shows the human user the requested information. The visualization uses different types of visualization techniques that include column charts, line charts, tables, indented tree layout [9] and organization charts. All of these techniques are common and well known, and they are chosen on the basis of the conclusion in appendix A.4.

The prototype has buttons that allow the human user to manipulate how the visualizations show the information. The human user can modify how information is used for the visualization. The human user can add information to an existing visualization. The human user can change which information is used for the visualization, this will create a new visualization based on the new information. The human user can filter out information. One filter is based on making the visualization show information for a specific period. The human user can filter information based on the dataset composition, and only show parts of the dataset. Some visualizations allow the human user to change the unit of numbers, for example if some information is shown in bytes, the unit can be changed to kilobyte or megabyte.

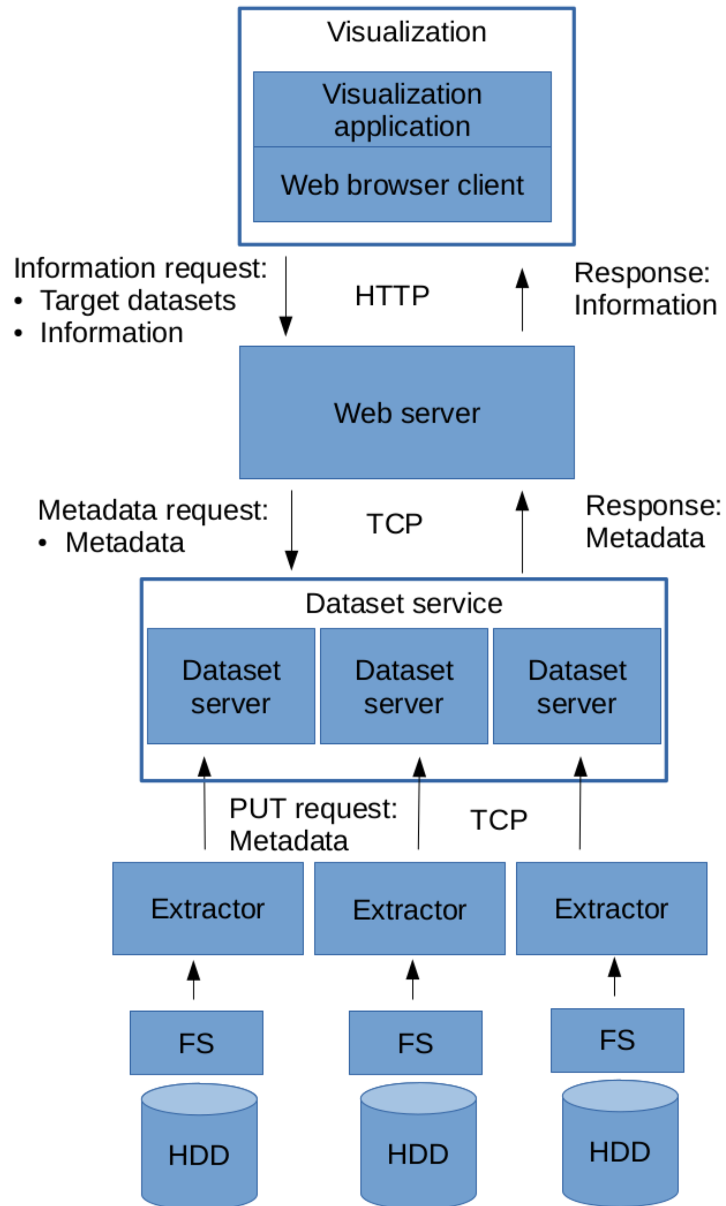


Figure 5.1: System design

5.1.1 Visualization application

The visualization application runs in a web browser client. The application creates the visualizations that are displayed to the human user. The visualizations are created with information that the application gets from the web browser client.

The visualization application uses buttons to allow the user to give input that changes how the visualization application creates the visualization. Some operations such as filtering information is done by the visualization application. While operations such as requesting new or additional information is executed by the web browser client.

5.1.2 Web browser client

The web browser client executes the visualization application. The web browser client keeps the information that is currently used by the visualization application in memory. If the visualization application requests new information, the web browser client will request the new information from the web server.

The web server has a list of commands that the web browser client can use to request information from the web server. The requests for new information uses the REST method GET.

5.2 Web server

The web server is responsible for delivering web pages to the web browser client. The web server uses the client-server model and delivers web pages on request. The communication between the web server and the web browser client uses the HTTP protocol.

The web server provides a list of commands that the web browser client can use to request information from the web server. The list of commands that the web server provides include:

- Get one piece of dataset information
- Get dataset information over time
- Get dataset size over time

- Get a list of dataset items

When the web server receives a request to one of the commands, the request includes the target dataset and the information that the client requests.

One request for information can consist of several pieces of metadata. The web server transforms the information requests into several metadata requests. The **dataset service** has a list of commands that the web server can use to request metadata from the dataset service. The web server is aware of all the **dataset servers** that the dataset service include. The web server sends the metadata request directly to the target dataset server.

The web server waits until it receives all of the metadata responses from the dataset service. When the web server has all the metadata, it transforms the metadata into the information that the web browser client requested and returns it to the web browser client.

5.3 Dataset service

The dataset service is comprised of multiple dataset servers. Each of the dataset servers provide a list of commands that clients can use. For this prototype the clients are:

- The web server - requests metadata from the dataset service
- The metadata extractor - adds metadata to the dataset service

The list of commands that the dataset service provides includes:

- Get metadata about an element in the dataset
- Get metadata about the dataset
- Get metadata about an element in the dataset for a period of time
- Get a list of elements in the dataset
- Add metadata about an element in the dataset
- Add metadata about the dataset
- Add metadata about an element in the dataset for a period of time

- Add a list of elements in the dataset

5.4 Dataset server

In this project the *dataset service* is realized with one *dataset server*. A dataset server include one unique dataset. Adding more dataset servers to the system gives us access to more unique datasets. But this approach would not scale the individual dataset server capacity.

The dataset server uses the client-server model with a custom protocol. The custom protocol uses stateless TCP connections. The dataset server uses a single thread. The dataset server executes commands sequentially. Executing commands sequentially makes each operation atomic. The dataset server stores the contained dataset in an in-memory database. The in-memory database uses data structures such as lists and sets to store metadata.

On request the in-memory database can save the dataset to disk. The in-memory database can be configured to save the dataset to disk on a set interval. This is not configured for this project, as the scope of the project didn't include handling failures and crashes.

5.5 Metadata extractor

The metadata extractor gathers metadata from a dataset. The dataset in this project is a filesystem. The metadata extractor iterates through the filesystem and gathers metadata about each file and directory. The metadata extractor has the following properties:

- Runs on a interval, set by a human user.
- Temporarily stores gathered metadata in memory.
- Add the temporarily stored metadata to a target dataset server.
  - Utilize commands provided by the dataset server.
  - Open a TCP connection to the target dataset server.
  - Bundle several commands into one TCP request.

6 Implementation

The implementation is shown in fig. 6.1.

6.1 Visualization

6.1.1 Technologies

The visualization uses HTML to create the web pages that contain the visualization application. The visualization application is created with JavaScript. The visualization application uses jQuery [1] to fetch information from the web server. The visualization application uses two different approaches for creating a JavaScript application.

React React [2] is an open-source project created by Facebook to build user interfaces. React is a JavaScript library that is based on encapsulated components. Since React is component based, all the JavaScript libraries that you want to use needs to have a React component version. These components can be developed by the original developer, or by an independent third-party developer. React is often used together with Redux [3], which is a predictable

1. http://jquery.com/
2. https://reactjs.org/
3. https://redux.js.org/

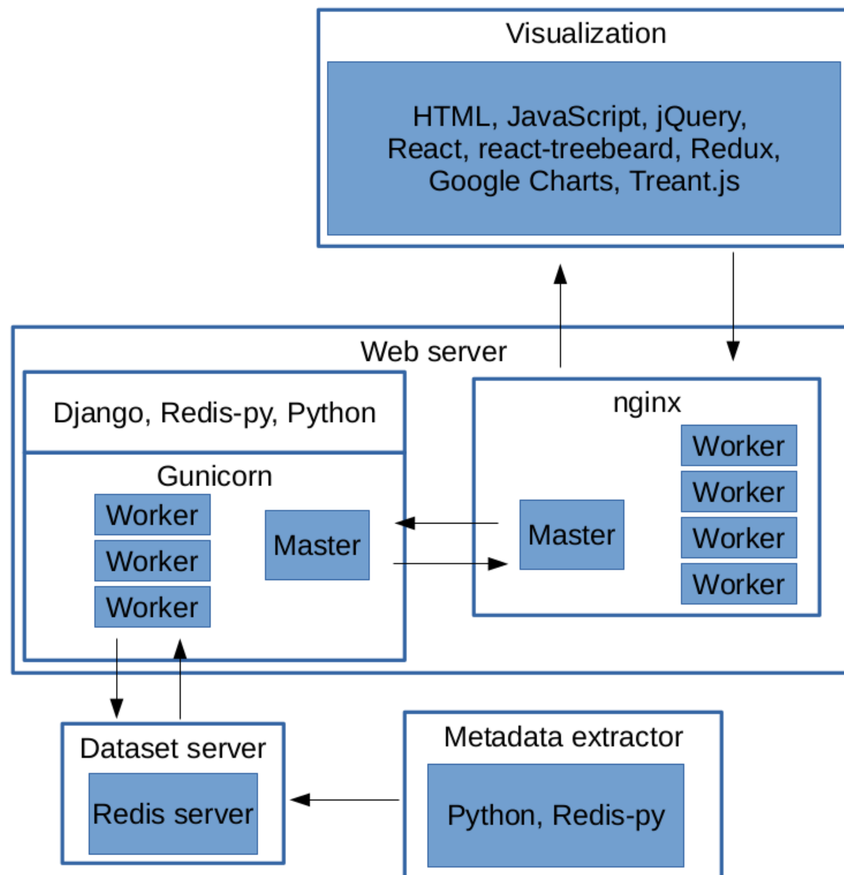


Figure 6.1: System implementation. Each blue square is a process.

state container for JavaScript. Redux is used in this prototype to handle state in the React application.

Pure JavaScript For this prototype some of the JavaScript libraries that we used, didn't have React components created by the original developer. One library had a React component created by a third-party developer and one library didn't have any React component. Creating parts of this prototype in pure JavaScript gave us the ability to directly interact with JavaScript libraries and use libraries without React components.

Google Charts One library that this prototype used extensively is Google Charts [4]. Google Charts is a visualization library for JavaScript developed by Google. It supports a rich amount of chart types. It has an extensive command list that gives the developer full control over each part of the chart creation. Google Charts also includes features such as controls and dashboards, that gives the human user control over specific parts of the chart.

react-google-charts [5] is a React component created by an independent developer. The component is a wrapper for the full Google Charts library. The difference between the React component and the native JavaScript library is discussed in section 9.6.

6.1.2 Information types

The visualization can request three types of information from the web server. Aggregate, metadata and historic.

Aggregate Aggregate information is one metric measured over time. An example of this is the total number of files. The command for this is "GET http://localhost/numfiles/". Aggregate information is retrieved as JSON. The requested metric has multiple entries with an associated value seen in listing 6.1.

```
{
    "2018-04-25 10:09:26": 516385,
    "2018-04-25 10:08:26": 513547
}
```

Listing 6.1: JSON response from total number of files command.

4. https://developers.google.com/chart/
5. https://github.com/RakanNimer/react-google-charts

Metadata Metadata is a collection of information about one directory. Metadata is the latest information that is extracted about a directory. The command for this is "GET http://localhost/metadata/*directoryname*/". Metadata is retrived as JSON where each directory has multiple fields with information. An example JSON response for the root directory is seen in listing 6.2


```
{
    "name": "./",
    "modified": "2018-04-21 12:02:43",
    "size": "121082556",
    "id": "9eb593bdb228c2a330ddeee74a"
}
```


Listing 6.2: Example JSON response from metadata command.

Historic Historic information is a collection of metadata for one directory. Historic information from one directory contains metadata for each collection time. The command is "GET http://localhost/history/*directoryname*/". Historic information is retrieved as a list of JSON, where each entry is a complete entry of metadata information. The collection time of each metadata is located within the metadata. An example JSON response for the root history is seen in listing 6.3. The collection time is called "m_time".


```
{
    "0": {
        "name": "./",
        "modified": "2018-04-21 12:02:43",
        "size": "121082556",
        "id": "9eb593bdb228c2a330ddeee74a",
        "m_time": "2018-04-25 10:08:26"
    },
    "1": {
        "name": "./",
        "modified": "2018-04-21 12:02:43",
        "size": "121082556",
        "id": "9eb593bdb228c2a330ddeee74a",
        "m_time": "2018-04-25 10:09:26"
    }
}
```


Listing 6.3: Example JSON response from history command.

6.1.3 Directory names

A standard Linux directory path is "/home/user/Documents/". The system uses relative path names starting at the root directory. This would create the command "GET http://localhost/metadata/home/user/Documents/". The web server parses that command as a complete URL and that is not a valid URL for the command "http://localhost/metadata/".

The system uses a default string name for the initial command. The name "root" is a valid string to request information about the root directory. The root directory contains information about the other directories. All other directory names are hash values, with the command "GET http://localhost/metadata/*hash*".

6.2 Web server

The web site is created in Python with the web framework Django [6]. Django is a high-level framework that focuses on rapid development. During development the prototype web pages was delivered by the Django lightweight development server. The development server is not design for anything else than serving as a simple development server. When we did experiments for this dissertation we deployed the web site to a production environment. The production environment uses **Gunicorn** and **nginx**.

Gunicorn [7] is a web server that is compatible with various web frameworks and uses the WSGI [14] calling convention. WSGI has two sides, the server/gateway side which talks to a reverse proxy or load balancer. The application/framework side which is compatible with Python frameworks. Gunicorn has one master process and several worker processes. The master process delegates work to the worker processes.

Gunicorn suggests [15] to use nginx [8] as a reverse proxy and web server that faces the client side. Nginx is a open-source web server that can be used as a reverse proxy, load balancer and HTTP cache. Nginx forwards all requests from the web browser client to Gunicorn. Nxinx has one master process and several worker processes. The master process delegates work to the worker processes.

6. https://www.djangoproject.com/
7. http://gunicorn.org/
8. http://nginx.org/

6.2.1 Commands

The Django framework defines commands that the Gunicorn web server offers to web browser clients. The list of commands that the web server offers includes:

- numfiles - Total number of files over time
- numfiletypes - Number of files of each file format
- fthistory - Average file format size over time
- historicsize - Size of the dataset over time
- metadatada - Directory metadata
- history - Directory metadata over time
- files - List of files in current directory
- subfolder - List of subdirectories in current directory

Some of these commands include the name of the requested directory. The directory name is either a string or a hash value, as detailed in section 6.1.3. When the web server receives one of these commands it transforms the command into one or several dataset server commands. For example the metadata history command uses three dataset server commands:

1. Get plain string name from hash value
2. Get list of historic metadata measurments
3. Get metadata for each list element

6.2.2 Response

The web server waits for all the responses from the dataset server. Each of the metadata responses from the dataset server are formated as bytestrings. The web server converts the bytestrings into integers, datetime objects and strings. The web server combines all the converted metadata into one JSON object and sends the JSON object as a response to the web browser client.

6.3 Dataset server

The dataset server is realized using Redis [9]. Redis is a single-threaded process. Redis has two main components, the Redis server and the Redis in-memory database. The Redis server allows clients to communicate with the Redis in-memory database. The Redis server uses a custom protocol utilizing TCP. The Redis database is realized as a data structure store. This means that Redis natively supports many foundational data structures and provides a rich set of commands for manipulating these data structures.

We chose Redis for this project specifically for the data structure support. Other alternatives could have been TimesTen [10] which is an in-memory relational database. But this would have given us a lot of functionality that the system doesn't utilize. Another alternative is Memcached [11] which is an in-memory key-value store. But this wouldn't have given us enough data structures to logical map the metadata that we want to store.

6.3.1 Commands

The dataset server in this project has two types of clients: the web server and the metadata extractor. The dataset server provides a rich set of commands that allow clients to manipulate the data types stored in Redis. The list of commands that the dataset server provide includes more advanced commands that we don't use. The advanced commands include comparing elements of the dataset and is discussed further in section 9.2.

The web server requests metadata from the dataset server, and uses these commands:

- get - returns the value of a key
- zrange - returns the specified range of elements in a sorted set.
- smembers - returns all members of a set.
- hgetall - returns all fields and values of a hash.
- lrange - returns the specified range of elements in a list.

9. https://redis.io/
10. http://www.oracle.com/technetwork/database/database-technologies/timesten/overview/index.html
11. https://memcached.org/

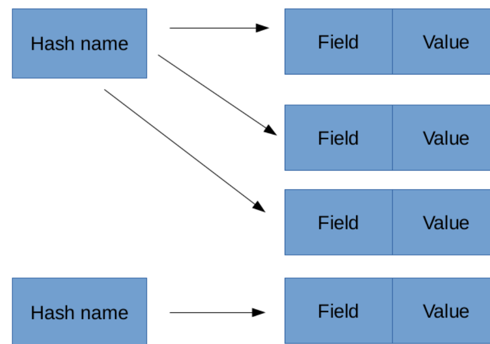


Figure 6.2: Redis hash

The metadata extractor adds metadata to the dataset server, and uses these commands:

- set - add value to a key.
- zadd - add an element with a score to a sorted set.
- lpush - add value to the head of the list.
- hmset - add field and value to a hash.
- sadd - add a member to a set.

6.3.2 Redis data structures

Redis provides the following data structures: key-value, hashes, sets, sorted sets and lists [16].

Key-value Key-value is a plain (key, value) combination. Only used to get the total number of directories.

Hash A hash is a collection of field-value pairs for a key. The key "Hash name" has several fields associated with it. See fig. 6.2. The system uses hashes to store metadata about a directory.

Set A set is an unordered collection of unique strings. The system uses sets to store names of all files in a specific directory.

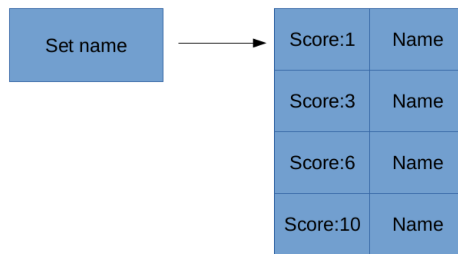


Figure 6.3: Redis sorted set

Sorted set A sorted set is a set that is sorted based on a score. See fig. 6.3. The system uses sorted sets to store information that either is sorted by date or by value.

When adding items to a sorted set the command is *zadd*. The Redis-py library changed the order of the arguments. Redis-py expects (name, score). The official Redis command expects (score, name).

List Lists are implemented as linked lists. This means that adding a new element to the head or tail is performed in constant time. The system only adds elements to either head or tail. In a linked list access by index is slower than lists based on arrays. In this prototype a list is only accessed sequentially and not by index. The system uses lists to store hashes from previous measurement dates.

6.3.3 Redis pipelines

Redis uses the request/response protocol. Every command sent to Redis creates a new request/response. Pipelining [17] is a technique to bundle several commands into one request/response. For example the client can create five *sadd* commands and add all five commands to a pipeline.The pipeline is sent to the server as one request. The server reads all five *sadd* commands and atomically performs each command. The server creates five responses and the five responses are put in the pipeline. The pipeline is sent as one response.

6.4 Metadata extractor

The metadata extractor prototype is written in Python 3.6. It iterates through every file and directory in the file system.

6.4.1 File system iteration

os.walk() is a built-in Python function in the os library. The function walks through every directory in the hierarchy.

```
for root, dirs, files in os.walk('./'):
```

Listing 6.4: os.walk() function

In listing 6.4 the current directory is supplied to the os.walk function. The os.walk function returns a three-tuple (root, dirs, files). *root* is a string path to the directory that os.walk currently resides in. *dirs* is a list of names for all subdirectories in the root directory. *files* is a list of names for all files in the root directory.

6.4.2 Metadata extraction

Directory metadata The directory metadata for each directory is added to a Redis hash. Each field in the hash is either metadata or a key to other data structures with directory information. The directory hash include the string name, modified date, directory size, number of subdirectories and number of files. It also include a unique hash ID, key to the filename set and the string name and hash ID of the subdirectories.

Directory size The size of each directory is reported by the *du* (disk usage) system call. This reports the size of the directory including all subdirectories. The directory size is added to the metadata hash.

Modified date The modified date for each directory is supplied by the Python function *os.path.getmtime()*. The function returns a timestamp that the system can transform into a local date with the Python function *datetime.datetime.fromtimestamp()*. The modified date is added to the metadata hash.

File formats For each file we get the file ending of the file. The file ending is defined as everything after the last period. This means that if a file has no period in the filename, the whole filename will be defined as the file format. The file ending is checked against a file format dictionary. A new entry is created if the file format does not exists in the dictionary. If the file format exists in the dictionary the entry is incremented. The dictionary is added to Redis as an sorted set where the name is the file format and the score is the number of occurrences.

Dataset size The dataset size is calculated by adding the size of each file. The dataset size is added to Redis as an sorted set where the name is the dataset size and the score is the date.

File names All filenames that are in a directory is added to Redis in a set.

6.4.3 Unique hash ID

The systems uses a hash ID to identify each directory and to identify the filename sets. The hash operation uses the *hashlib* Python library to create a hex value with SHA224 encryption. See the function in listing 6.5. The SHA-224 encryption was chosen because it creates the shortest string of the SHA2 encryptions. SHA-224 string length is 57 characters compared to SHA-256 string length of 65 characters. The encryption is not important except for creating a theoretical big enough key space. SHA-224 supports $2^{64} - 1$ keys.

```
name = plain_string_name
hash = hashlib.sha224(name).hexdigest()
```

Listing 6.5: hash function

6.5 Technologies

Python 2.7, Python 3.6, Django 1.11.6, Redis 4.0.2, Gunicorn 19.7.1, nginx 1.10.3.

npm 5.5.1, React 16.0, Redux 3.7.2, React-Redux 5.0.6, Treebeard 2.1, React-Google-Charts 1.5.5. jQuery 3.3.1, Treant.js 1.0, Redis-py 2.10.6.

The hardware and OS is listed in table 6.1.

6.6 File creation

To check that the system works as intended we need a structured method to create files and directories. In Python the *os* library has functions that the prototype used:

- chdir() - change directory

Hardware		
	Vendor	Lenovo
	CPU	Intel Core i5-6400T @ 2.20GHz
	Ram	16 GiB
	Hard disk	ATA model MZ7LN512HMJP
Operating system		
	OS	Mint 18.2 ("Sonya")
	Kernel	Linux 4.10.0
Filesystem		
	Test partition	495 GB Linux filesystem
	Filesystem	ext4
	I/O Scheduler	noop deadline cfq

Table 6.1: Computer specifications

- mkdir() - create a new directory
- remove() - remove a file

The prototype also used the Python file object. The file object has the following functions:

- open() - create a new file if it does not exist
- write() - writes a string or bytes to the file
- close() - close the file

The file creation prototype pseudo code can be seen in listing 6.6. The prototype creates a new directory and file in the current directory. Inside the new directory it creates ten files. It does this forever. After ten iterations the prototype changes the current directory to one of the new directories that was created.

```
import os
i = 10
k = 0
while(True):
        create_file(unique_filename)
        new_directory = create_directory(unique_directoryname)
        change_to_new_directory()
        for range(0, i):
                create_file(unique_filename)
        change_to_previous_directory()
        k = k + 1
```

```
        if k > 10:
            change_current_directory()
```

Listing 6.6: File creation

Experiments

The experiments are focused on the four distinct parts of the system and the system as a whole. We include two experiments that highlights potential issues. This section will outline what we focus on for each part and define the metric we use for the experiments. The next section explains the benchmarking tools that we use for the experiments. The following sections details each of the experiments.

All experiments are executed on the same computer, listen in table 6.1.

Metrics

- Memory utilization - the physical memory that the target process has allocated.
- CPU utilization - the percentage of capacity on a single CPU core a target process uses.
- Requests per second - the amount of network requests a target process sends and gets a response to per second.
- Latency - the round-trip time. The round-trip time is the time in milliseconds from a request is sent from a client, the server processes the requests and sends a response, and the client receives the response.

- # requests - total number of request sent by a target process.
- Elapsed time - the time in seconds a target process uses from a start point to a stop point.
- Data transferred - the amount of bytes the target process has either sent or received.

Web browser client For the web browser client we measure the memory footprint, # requests, elapsed time and data transferred.

Web server For the web server we measure the CPU utilization of the two web servers, Gunicorn and Nginx. We want to measure the requests per second and latency for the two web server together.

Dataset server For the dataset server we measure the requests per second, CPU utilization and memory utilization. We also want to analyze the keys in the dataset.

Metadata extractor For the metadata extractor we measure the CPU utilization, memory utilization and the elapsed time of each extraction.

System We measure the CPU utilization and memory utilization of all the parts of the system when everything executes at the same time.

os.walk We measure the execution time of the os.walk function in both Python2.7 and Python3.6. Python3.6 includes a new implementation of the function.

Reported disk usage We measure the reported disk usage of the GUI file explorer Nemo, the system call du and manually calculating the disk usage with os.walk.

7.1 Benchmarking tools

7.1.1 wrk

The program *wrk* [1] is used to benchmark the web server. *wrk* is an open source HTTP benchmarking tool. It generates load for the web server. Wrk send a

1. https://github.com/wg/wrk

```
Running 5s test @ http://localhost/
  2 threads and 100 connections
  Thread Stats   Avg      Stdev     Max   +/- Stdev
    Latency    32.43ms    2.76ms  44.68ms   86.70%
    Req/Sec     1.54k    93.12     1.68k    77.00%
  15374 requests in 5.00s, 6.89MB read
Requests/sec:   3071.86
Transfer/sec:      1.38MB
```

Figure 7.1: wrk sample output

request to the url and waits for the url to execute the request, and receives the response. This means that wrk measures the round-trip time of a request. The *wrk* command is given in listing 7.1:

```
$ wrk -t4 -c10 -d60s --timeout 15s "URL"
```

Listing 7.1: wrk command

The command line options used for this benchmark are:

- -t : Number of threads, default 4
- -c : Number of concurrent HTTP connections, no default
- -d : Duration, default 60 seconds
- –timeout : Timeout (how long each requests waits before timeout error), default 15 seconds

The number of concurrent HTTP connections is the option we change for the benchmarks. All experiments uses a duration of 60 seconds unless otherwise stated in the experiment.

The output from *wrk* can be seen in fig. 7.1. The relevant metrics for our experiments are:

- Average latency
- Standard deviation
- Requests per second

7.1.2 psutil

psutil (process and system utilities) [2] is a cross-platform Python library for retrieving information on running processes and system utilization. It natively implements functionality offered by UNIX command line tools.

CPU usage *psutil* has a function *cpu_percent* that returns a float representing the process CPU utilization as a percentage. The percentage can be more that 100.0 in the case of a process running multiple threads on different CPUs. The cpu_percent function is used on independent target processes with the parameter *interval=1*. This means that the psutil process will monitor the target process for one seconds and report the cpu utilization.

memory utilization *psutil* has a function *memory_info* [18] that returns a named tuple with variable fields. One field is *rss* aka “Resident Set Size”. *rss* is the non-swapped physical memory a target process uses.

7.1.3 Redis benchmark

Redis includes a benchmark utility, redis-benchhmark [19]. The redis-benchmark simulates running commands done by a number of clients at the same time, sending a total number of queries. The tool sends requests to the Redis server and waits for a response. Measuring the round trip time. Before the benchmark is started the database is flushed. The redis-benchmark command can be seen in listing 7.2.

```
$ redis-benchmark -q
```

Listing 7.2: Redis benchmark command

The *-q* option is for running the benchmark in quite mode, this only shows the query per seconds values. The default values for redis-benchmark is to create 50 parallel connections to the Redis server. The total number of commands are 100,000.

7.2 Experiment dataset

The experiments needs datasets to extract metadata from. For the experiments the main dataset is a selection of COAT directories. The directories are "fotoboks2011" and "fotoboks2012".

2. https://github.com/giampaolo/psutil - version 5.4.5

	Big	Medium	Small
Size	122.93 GB	5.05 GB	0.41 GB
Files	513,550	18,058	236
Directories	126	2252	48

Table 7.1: Dataset differences

- fotoboks2011 - 249,691 items, totaling 59,7 GB
- fotoboks2012 - 263,978 items, totaling 63,3 GB
- total - 513,551 files in 127 directories. Totaling 122 GB

This dissertation uses three different datasets and they are given a common identifying name, defined in table 7.1. The two COAT directories are classified as a Big dataset. The Big dataset consist mostly of .jpg files. The experiments uses the "Documents" directory of the experiment computer, as a Medium dataset. The Medium dataset consist of an arbitrary mix of file formats and file sizes. The Small dataset is created by a consistent loop by the file creation prototype. All files in the Small dataset are one mb in size.

7.3 Web browser client

The experiment uses two different web browser clients: Google Chrome and Mozilla Firefox. The measurements are measured by the developer tools included in both web browsers [20] [21].

For this experiment the web server processes and the dataset server is running. The metadata extractor is not running during the experiment. Before the experiment we run the metadata extractor 16 times, this means that there are 16 historic metadata hash tables. The dataset is the Big dataset defined in table 7.1.

The experiment measures three different web pages. The three different web pages uses different technology and libraries.

- Home - uses React, Redux and Google Charts.
- /chart - uses Google Charts.
- /overview - uses Treant.js.

7.4 Web browser client - memory footprint

7.4.1 Methodology

The experiment measures the memory footprint of the web browser client when visiting different web pages. The developer tools includes a memory snapshot tool which reports the total memory footprint of the current page. The snapshot of the memory footprint is taken after a page is finished loading.

7.4.2 Metrics

- Firefox memory footprint
- Chrome memory footprint

7.5 Web browser client - network usage

7.5.1 Methodology

The experiment measures the network usage of the web browser client when visiting different web pages. The developer tools include a network panel which gives insight into resources requested and downloaded over the network. The browser cache is disabled for all measurements.

7.5.2 Metrics

- # requests
- Elapsed time
- Data transferred

7.6 Web server

The performance metrics measured for the web server are:

- Requests per second for web pages

- Latency for web pages
- Requests per second for commands
- Latency for commands
- Gunicorn CPU utilization
- Nginx CPU utlization

For all the web server experiments the web server and the dataset server is running. The metadata extractor is not running during the experiment. Before the experiment we run the metadata extractor 10 times, this means that there are 10 historic metadata hash tables. The experiments use the Big dataset defined in table 7.1.

NOTE: the experiment is done on localhost, this means that the web server and the measurement tool is running on the same physical machine.

Web pages The web server delivers three web pages that we want to examine. Each of the web pages requests different types of information.

- Home page - requests two sorted sets from the dataset server.
- /chart - requests one metadata hash from the dataset server.
- /overview - requests the current metadata hash for all directories.

Commands The web server has a list of commands that requests metadata from the dataset server.

- history - gets a list of historic data for a target directory. For each entry in the list, gets the metadata for the specific measurement
- metadata - gets metadata for a target directory
- files - gets a set with all filenames for a target directory
- historicsize - gets a sorted set with historic measurements of the dataset size
- numfiles - gets a sorted set with historic measurements of the total number of files in the dataset

7.7 Web server - requests per second

7.7.1 Methodology

The *wrk* command (section 7.1.1) is used to create GET requests for the web pages and the commands. The wrk command is run five times for each number of connections. The number of connections are 10, 20, 30, 40, 50. We parse the output from wrk to get the *Requests/sec* number.

7.7.2 Metrics

- Home page requests per second
- /chart page requests per second
- /overview page requests per second
- history command requests per second
- metadata command requests per second
- files command requests per second
- historicsize command requests per second
- numfiles command requests per second

7.8 Web server - latency

7.8.1 Methodology

The *wrk* command (section 7.1.1) is used to create GET requests for the web pages and the commands. The wrk command is run five times for each number of connections. The number of connections are 10, 20, 30, 40, 50. We parse the output from the wrk command to get the average latency and the standard deviation numbers.

7.8.2 Metrics

- Home page latency and standard deviation

- /chart page latency and standard deviation
- /overview page latency and standard deviation
- history command latency and standard deviation
- metadata command latency and standard deviation
- files command latency and standard deviation
- historicsize command latency and standard deviation
- numfiles command latency and standard deviation

7.9 Web server - CPU usage

7.9.1 Methodology

The CPU usage measurements are done by using psutil (section 7.1.2) to get the cpu_percent number for each process. The pseudo code for the measurements can be seen in listing 7.3.

```
import subprocess, psutil, time

command = ["pgrep", "-f", "name"]
process_pid = subprocess.check_output(command)
cpu_usage = []
time = 0
start = time.time()
while(elapsed < 60 seconds):
        stop = time.time()
        elaspsed = float(stop - start)
        curr_cpu = process_pid.cpu_percent(interval=1)
        cpu_usage.append(curr_cpu)
```

Listing 7.3: psutil capture of CPU usage

To create load for the web server we use the *wrk* command (section 7.1.1). We check the CPU usage for 10 and 50 concurrent connections. We check the /chart page and the /overview page.

Gunicorn Gunicorn runs four processes. One parent process, which only does light work distribution. Three worker processes that does all computation. We get the CPU utilization for all four processes. This means that one process should use close to 0% CPU at all times.

Nginx Nginx runs five processes. One parent and four workers. We get the CPU utilization for the workers.

7.9.2 Metrics

- Gunicorn CPU utilization
- Nginx CPU utilization

7.10 Dataset server - keys

7.10.1 Methodology

Redis includes a key space analyzer. The key space analyzer command can be seen in listing 7.4.

```
$ redis-cli --bigkeys
```

Listing 7.4: Redis bigkeys command

The key space analyzer provides information about the data stored in Redis. The information includes which data types are used and the size of each of the data types. It also provides some aggregate information, including total number of keys. It gives us the total and average key length in bytes. The key space analyzer is run on a Redis dataset that does not change.

To measure the size of individual keys we use a script found on Github which is included in Appendix B. It uses built-in Redis functionality and CLI commands to list all keys and their size.

This experiment uses the Big dataset defined in table 7.1. Before the experiment we run the metadata extractor 24 times, this means that there are 24 historic metadata hash tables.

7.10.2 Metrics

- Number of keys
- Type of keys
- Size of keys

7.11 Dataset server - memory utilization

7.11.1 Methodology

The experiment measures the memory utilization of the in-memory database. To measure the memory utilization we must add keys to the database. Adding 10,000 keys to Redis over TCP is slowed down by the need to use a TCP connection. Redis has a tool which pipes commands directly to Redis. This allows commands to go directly to Redis. This is designed for mass insertion operations [22].

By creating a large number of similar keys, we can determine the growth of memory utilization based on the number of keys. This is a synthetic test without any of the data structures that the system actually uses.

We create a small Python script that can create a file with a number of commands. The commands are "SET *unique key - value*". The script creates five files with increasing number of commands.

- 10,000
- 100,000
- 1,000,000
- 10,000,000
- 100,000,000

The command files is then given to a mass insertion Python script. The mass insertion script is found on github, and can be seen in appendix C. The script transforms the basic commands into Redis command strings. The output from the mass insertion script is piped into Redis, see command 7.5.

```
$ python redis-mass.py input.txt | redis-cli --pipe
```

Listing 7.5: Mass insertion command

7.11.2 Metrics

- Redis memory utilization

7.12 Dataset server - CPU utilization

7.12.1 Methodology

The CPU utilization measurements are done by using psutil (section 7.1.2) to get the cpu_percent number for the Redis server process. The pseudo code for the measurements are the same as used for the web server and can be seen in listing 7.3.

For this experiment we use the Big dataset defined in table 7.1. We create two scenarios where we measure the CPU utilization.

Idle The Idle status of the system is that the web server, dataset server and the metadata extractor are running.

Load The web server, dataset server and the metadata extractor are running. To create load for the dataset server we run the redis-benchmark utility (section 7.1.3). The redis-benchmark utility is run with the default options, 50 parallel connections and total 100,000 commands. The redis-benchmark is started manually from the command line. We wait 10 seconds before we start the redis-benchmark. It runs for about 30 seconds.

7.12.2 Metrics

- Dataset server CPU usage Idle

- Dataset server CPU usage Load

7.13 Dataset server - requests per second

7.13.1 Methodology

The experiment measures the requests per seconds for the dataset server. To measure specifically the requests per second for the Redis server, we use the redis-benchmark utility (section 7.1.3). The redis-benchmark is run with the command seen in listing 7.6.

```
$ redis-benchmark -q -r 100000 -n 100000
```

Listing 7.6: Redis benchmark command

The *-n* specified the number of commands to perform. The *-r* specifies the keyspace. Setting the keyspace to the same number as the commands means that we get a unique key for every command. The default is to use the same key for every command.

7.13.2 Metrics

- Requests per second for each command

7.14 Metadata extractor - execution time

7.14.1 Methodology

This experiment measures the metadata extractor execution time. The metadata extractor is run on a root directory without any changes to the directory structure. This is because we don't want any extra CPU usage because of dataset changes. The experiment will use three different datasets, Big, Medium and Small. The datasets are defined in table 7.1.

The metadata extractor reports statistics for each iteration, the metadata extractor code can be seen in listing 7.7. For this experiment we save the elapsed time for ten iterations and then plot the results in a separate program. This is to get three different root directories in the same plot.

```
import time
start = time.time()
exctract_all_information()
send_information_to_redis()
stop = time.time()
```

```
elapsed = float(stop - start)
```

Listing 7.7: Metadata extractor execution time

7.14.2 Metrics

- Execution time

7.15 Metadata extractor - resource usage

7.15.1 Methodology

This experiment measures the resource usage of the metadata extractor. We use psutil (section 7.1.2) to measure the resource ussage. The metadata extractor runs for 1 minute, with 5 seconds delay between each iteration. The pseudo code implementation can be seen in listing 7.8. Note that the memory_info().rss function returns the memory utilization in bytes, and we convert it to mb.

To avoid extra memory utilization or CPU load the experiment is run on a static dataset. The three dataset are defined in table 7.1. The Redis database is flushed before the first iteration.

```
import time, subprocess, psutil

pid = extractor_pid
process = psutil.Process(pid)
cpu_array = []
mem_array = []
start = time.time()
elapsed = 0
while(elapsed < 60 seconds):
        stop = time.time()
        elapsed = float(stop - start)
        cpu = p.cpu_percent(interval=1)
        mem = p.memory_info().rss
        mem = mem / 1000000 #get mb
        cpu_array.append(cpu)
        mem_array.append(mem)
```

Listing 7.8: Metadata extractor resources usage

7.15.2 Metrics

- Metadata extractor memory utilization
- Metadata extractor CPU usage

7.16 System - Resource usage

7.16.1 Methodology

This experiment measures the resource usage of all parts of the system. The two main resources that the system uses is CPU and memory. There are four distinct prototype parts that we can measure. The prototype parts are:

- Metadata extractor
- Dataset server
- Gunicorn
- Nginx

The metadata extractor and the dataset server both use one process. Gunicorn has three worker processes, for the CPU measurements we use the average of the three individual processes. Nginx has four worker processes, for the CPU measurements we use the average of the four individual processes.

The CPU usage measurements are done by using psutil (section 7.1.2) to get the cpu_percent number for each process. The memory measurements are done by using psutil to get the rss number for each process. This experiment uses the Big dataset that is defined in table 7.1.

We measure the state of the system in two different scenarios. For both scenarios the dataset server and the web server is running.

Idle The Idle status of the system is that the web server processes, dataset server and the metadata extractor is running. The metadata extractor runs with a delay of ten seconds between iteration, this is to get a better view of when it's running and not.

Load The system status is identical to the Idle status. To create a load on the system we use wrk (section 7.1.1). We measure both with 10 and 50 concurrent connections. We use the /chart web page as the target for this experiment. The /chart page involves all of the parts of the system.

We start wrk manually from the command line. wrk stops after 60 seconds, then we wait for 10 seconds before we start it again. The delay is because we want to identify when it stops in the graphs.

7.16.2 Metrics

- System CPU usage Idle
- System CPU usage Load
- System memory utilization Idle
- System memory utilization Load

7.17 os.walk

7.17.1 Methodology

The experiment measures the execution time of the function os.walk. We want to measure the the execution time in both Python 2.7 and Python 3.6. The experiment code is identical for both Python versions and is listen in listing 7.9.

The measurements uses the Big dataset, defined in table 7.1, as the root directory. The measurements are run 10 times and we measure the execution time for each iteration.

7.17.2 Metrics

- Execution time for Python 2.7
- Execution time for Python 3.6

```
import os, time
iterations = 10
```

```
for range (0, iterations):
        test_time, num_files, num_dirs = run_test()
        cummulative_time = cummulative_time + test_time
average_time = cummulative_time / iterations

def run_test():
        num_files = 0
        num_dirs = 0
        start = time.time()
        for path, dirs, files in os.walk('./'):
                num_files = num_files + int(len(files))
                num_dirs = num_dirs + int(len(dirs))
        stop = time.time()
        elapsed = float(stop - start
        return elapsed, num_files, num_dirs
```

Listing 7.9: os.walk experiment

7.18 Reported disk usage

7.18.1 Methodology

The experiment measures the reported disk usage of a directory including subdirectories. We want to measure the reported disk usage of three different utilities: os.walk, du -s and Nemo. os.walk is a Python function that iterates through all directories and allows us to measure the disk usage of each directory and file. du -s [23] is a Linux system call that calculates the disk usage of the current folder. Nemo [24] is a GUI file explorer that is used in Linux Mint.

The primary goal of this experiment is to identify differences between os.walk and du -s. Nemo is included as a reference point for the measurements.

The experiment is run once per directory. The calculations does not change as long as there are no changes to the files or directories. The experiment is run on three different directories. Each of the directories include different directory structures and a mix of file formats. The composition of the directories are discussed with the result.

7.18.2 Metrics

- Reported disk usage for: os.walk

- Reported disk usage for: du -s
- Reported disk usage for: Nemo

```
import os, subprocess
from os.path import join, getsize
walk_size = 0
for path, dirs, files in os.walk('./'):
        for name in files:
                file_path = join(path, name)
                file_size = getsize(file_path)
                walk_size = walk_size + file_size
        for name in dirs:
                dir_path = join(path, name)
                dir_size = getsize(dir_path)
                walk_size = walk_size + dir_size
du_size = subprocess.check_output(['du','-s', path])
du_size = du_size.split()[0].decode('utf-8')
```

Listing 7.10: Disk usage measurement

8 Results

All of the raw experiment data is available in the source code zip file, in the "benchmarks/raw_data" directory.

8.1 Web browser client - memory footprint

Table 8.1 shows the memory footprint of two web browsers, Chrome and Firefox. From the table we can see that the Home page uses 13.7 mb in Chrome and 9.3 mb in Firefox. The /chart page uses 10.1 mb in Chrome and 11.1 mb in Firefox. The /overview page uses 6.0 mb in Chrome and 6.9 mb in Firefox.

The memory footprint of the Home page and the /chart page can partly be explained by both web pages using the Google Charts library. The Google Charts library is a library with a lot of functionality. The Home page includes React and Redux, but from the memory footprint this does not seem to increase

Page	Chrome	Firefox
Home	13.7 mb	9.3 mb
/chart	10.1 mb	11.1 mb
/overview	6.0 mb	6.9 mb

Table 8.1: Web client memory footprint

	Requests	Time	Data transferred
Chrome			
Home	12	419 ms	2.1 mb
/chart	17	417 ms	432 kb
/overview	136	1.38 s	332 kb
Firefox			
Home	12	460 ms	2.13 mb
/chart	19	582 ms	446 kb
/overview	263	5 s	495 kb

Table 8.2: Web client network usage

the memory footprint significantly.

The /overview page uses a library with one main functionality, and a lot fewer functionalities than Google Charts. This results in the /overview page using less memory than the other two pages.

Considering that Firefox and Chrome uses over 100 mb of memory just by running, and is widely known to use many hundred mb of memory. Having a web page that uses 10-15 mb of memory doesn't impact the system negatively.

The raw experiment data is available in the file:

"benchmarks/raw_data/web_browser_client.ods".

8.2 Web browser client - network usage

Table 8.2 shows the network usage of the two web browsers, Chrome and Firefox. From the table we can see that the Home page and the /chart page create almost the same number of requests and uses the same amount of time for both web browsers. The interesting thing about the Home page and the /chart page is that the Home page downloads over 2 mb of data, while the /chart page downloads under 500 kb.

The difference between these two pages are that Home uses React and Redux. React and Redux in them self should be around 140 kb [25]. To create the React application the development environment is set up to use Webpack [1] as a bundler for all the JavaScript files and libraries. The default configuration of Webpack bundles all libraries into one "bundle.js" file. When Webpack creates

1. https://webpack.js.org/

the "bundle.js" file, it reports the size of the file. For this prototype the "bundle.js" file was 1.86 mb.

For development, Webpack bundles the libraries "as is". A common practice for JavaScript is to use minification [26], which is a process that removes all unnecessary characters from source code without changing the functionality. Webpack was configured to minify all the source files [20]. This reduced the size of the "bundle.js" from 1.86 mb to 449 kb. This results in the Home page reducing the amount of data it downloads from over 2 mb to 800 kb.

Looking at the /overview page in table 8.2, we can see that the amount of data downloaded is similar in both web browsers. Firefox uses double the amount of time compared to Chrome. The time is explained by the number of requests, and it seems like Firefox reports a 301 HTTP response for each request we send to the dataset server. The 301 HTTP response is "Moved Permanently" which indicates that it's a redirect. Chrome doesn't give a warning about the redirect. All the requests are redirected from Nginx to Gunicorn. The fact that Firefox gives us the warning points to something wrong with the configuration of Nginx.

In section 9.2 there is an discussion about how we could utilize Redis and improve the way we request information.

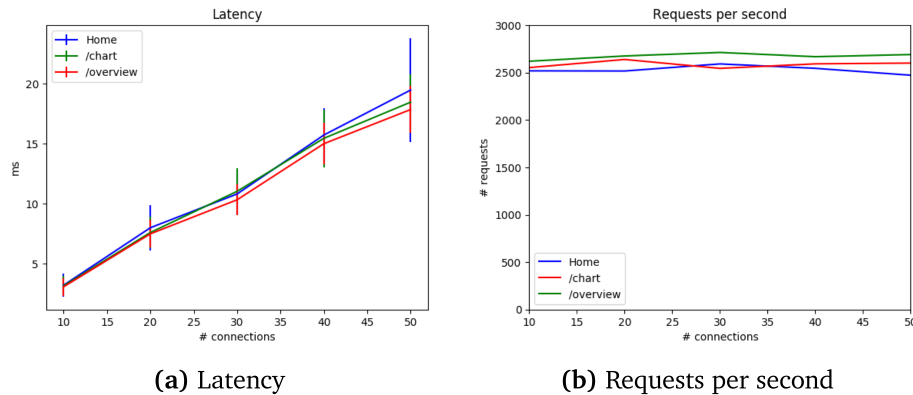
The raw experiment data is available in the file:

"benchmarks/raw_data/web_browser_client.ods".

8.3 Web server - web pages

The fig. 8.1 shows us the requests per second that the web server can deliver web pages to a client. The figure also shows the latency and standard deviation for the requests. From the figure we can see that all three web pages are similar for both metrics. Figure 8.1b shows us that requests per second is stable at around 2500 requests per seconds for all three web pages. The requests per second is stable for the different number of concurrent connections.

The fact that the requests per second is the same for 10 and 50 concurrent connections indicates that the web server has an upper limit of 2500-3000 requests per second for this prototype. If the upper limit for the web server was higher, we should have seen a higher number of requests per second for 10 concurrent connections, and then an decrease of requests per second as the number of concurrent connections increase.



(a) Latency

(b) Requests per second

Figure 8.1: Web server requests per second and latency for delivering web pages

This is most likely because of the configuration of Nginx and Gunicorn. For this prototype both of the web servers are configured with the default values. For example Nginx has a default value [27] of 100 requests a client can make over a single connection. This would directly impact this experiment as wrk sends a large number of requests from a single client.

Figure 8.1a shows us that the latency is similar for all three web pages. All three web pages has a similar increase of latency. All three web pages sees an increase in the standard deviation as the number of concurrent connections increase.

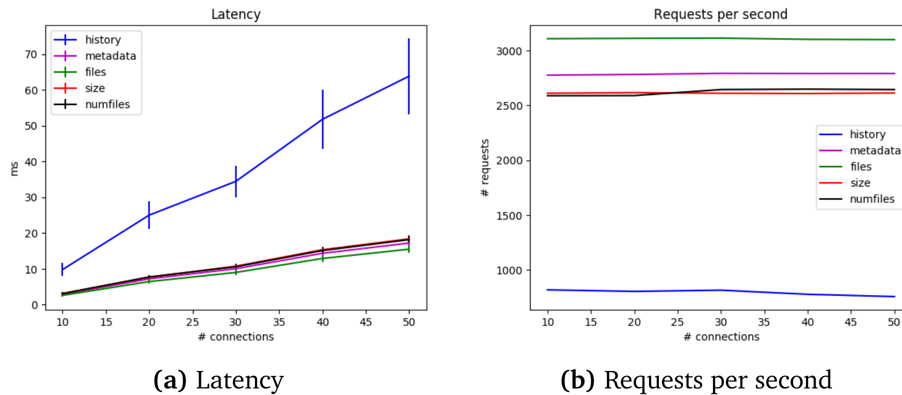
This experiment shows that the system delivers all web pages in a consistent way. There is no web page that is different from the others. This is because of the web server commands they use, and if we would have used the "history" command in one of the web pages, we would have seen a different result. As we will see in the "web server - commands" experiment.

The raw experiment data is available in the file:

"benchmarks/raw_data/web_server_data.py".

8.4 Web server - Commands

The fig. 8.2 shows us the requests per second that the web server can respond to a command request from a client. The figure also shows the latency and standard deviation for the requests. The web server commands are the list of commands that sends a request to the dataset server and returns the information



(a) Latency **(b)** Requests per second

Figure 8.2: Web server requests per second and latency for responding to commands

as JSON objects.

Figure 8.2b shows us that four out of five commands achieve 2500-3000 requests per second. From the figure we can see that the "history" command achieves fewer requests per second compared to the other commands. The "history" command services around 800 requests per second.

Figure 8.2a shows us the that four out of five commands has similar latency and standard deviation. At best the commands has a latency of 2.5 ms with a standard deviation of 0.3 ms. At worst the commands has a latency of 18 ms with a standard deviation of 1.1 ms. From the figure we can see that the "history" command has higher latency and standard deviation compared to the other commands. The "history" command at best has a latency of 9.7 ms with a standard deviation of 1.7 ms. At worst the "history" command has a latency of 63.8 ms with a standard latency of 10.6 ms.

The way the "history" command works is that it first requests a list of historic measurements. Then for each measurement in the list, it requests the metadata for that measurement. The list contains ten measurements, this means that each "history" command request is one request for the list and ten requests for metadata. Compared to all the other commands that sends one request to the dataset server.

Most of the commands are consistent with each other. But the "history" command is much worse both in terms of requests per second and latency. This means that any client that would use this command could have a negative experience. In section 9.2 there is an discussion about how we could improve the "history" command.

The raw experiment data is available in the file:

"benchmarks/raw_data/web_server_commands_data.py".

8.5 Web server - CPU utilization

Figure 8.3 shows us the CPU utilization of the Gunicorn processes. The master process is a blue line that is constant at 0%. The Gunicorn worker processes are at around 90% CPU utilization. The experiment computer has four cores, and each of the Gunicorn process uses 90% of each of the CPU cores. Figure 8.4 shows us the CPU utilization of the Nginx processes. We can see that the Nginx processes doesn't exceed 20% CPU utilization.

Gunicorn handles the communication between the web server and the dataset server. While Nginx only forwards the requests from the web browser client to Gunicorn. This explains some of the reasons why Gunicorn uses more CPU than Nginx. The measurements shows that Gunicorn uses most of the resources the computer can give to it, while Nginx has free capacity.

The raw experiment data is available in the files:

- "benchmarks/raw_data/gunicorn_history_10c.py"
- "benchmarks/raw_data/gunicorn_history_50c.py"
- "benchmarks/raw_data/gunicorn_metadata_10c.py"
- "benchmarks/raw_data/gunicorn_metadata_50c.py"
- "benchmarks/raw_data/nginx_history_10c.py"
- "benchmarks/raw_data/nginx_history_50c.py"
- "benchmarks/raw_data/nginx_metadata_10c.py"
- "benchmarks/raw_data/nginx_metadata_50c.py"

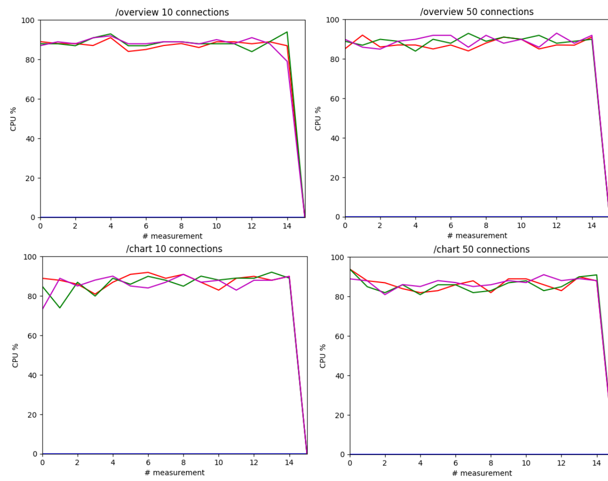


Figure 8.3: Gunicorn CPU utilization

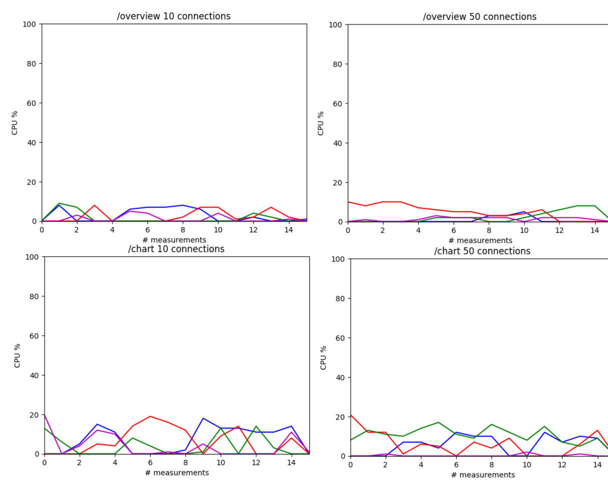


Figure 8.4: nginx CPU utilization

Data type	Items	Total size	Description
set	8500	107.91 kb	A set of filenames
hash	26	1.41 kb	Metadata for a directory
list	24	1 kb	List of historic measurements
zset	24	1 kb	Sorted set of dataset size over time

Table 8.3: Biggest data types

Data type	# data types	# items in data type	Average size
hash	3150	31475	10 b
list	126	3024	24 b
set	123	513548	4175 b
zset	8	173	22 b

Table 8.4: Aggregate data type information

8.6 Dataset server - keys

Table 8.3 shows the biggest data type of each data type. Sets are the biggest with 8500 items. This size will not change as we only have one set with the current set of filenames.

Table 8.4 shows aggregate information about each data type. Hashes are the interesting one here, as we will create a new hash for every directory for each iteration of the metadata extractor. The average size of each hash is 10 bytes.

Table 8.5 shows the summary for the dataset. 3407 keys uses a total of 149 kb.

The table shows that for 24 measurements the overall memory footprint is less than 1 mb. Considering that most modern computers at the time of writing this paper, has 16 GB of memory, 1 mb is very low. The only data type that uses more than a few byte is the sets with filenames, but the sets do not increase in size without adding more files to the dataset.

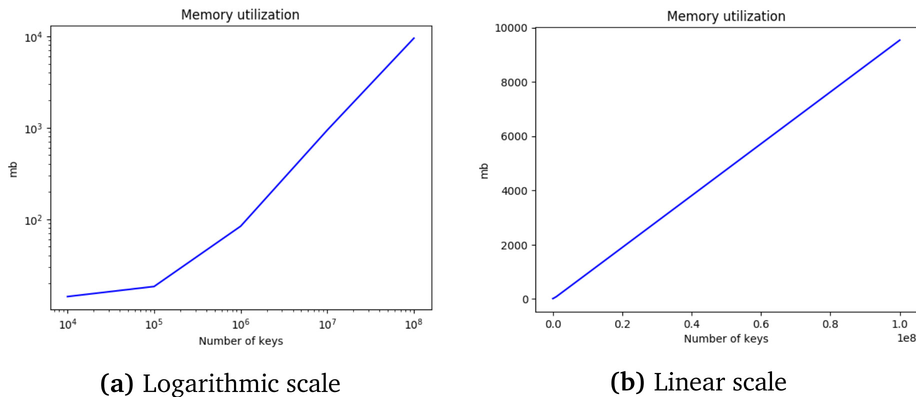
These measurements in these tables are discussed further in section 9.4.

The raw experiment data is available in the files:

- "benchmarks/raw_data/bigkeys_output.txt"
- "benchmarks/raw_data/redis_key_sizes.txt"

Total keys	3407	
Total size	149327 b	149 kb
Average size	44 b	

Table 8.5: Total keyspace information



(a) Logarithmic scale

(b) Linear scale

Figure 8.5: Dataset sever memory utilization

8.7 Dataset server - memory utilization

Figure 8.5 shows the memory utilization for an increasing number of keys. The number of keys measured begins with ten thousand and increases with an order of magnitude up to 100 million. The values are from 15 mb to 9 GB.

In fig. 8.5a both scales are logarithmic because the x-axis increases with an order of magnitude for each measurement.

In fig. 8.5b both scales are linear. Here we can see that the overall scaling of the memory utilization is linear.

The experiment shows us that the memory footprint increases linear with the amount of keys. There are no unexpected jumps in the memory footprint. This allows us to reason about and calculate the memory footprint of the system. It also shows us that 100 million key-value pairs uses 9 GB of memory, which is a good result.

The raw experiment data is available in the file:

"benchmarks/raw_data/dataset_server_keys_data.py".

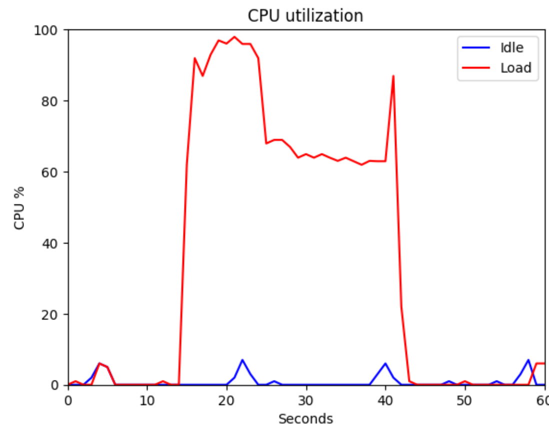


Figure 8.6: Dataset server CPU utilization

8.8 Dataset server - CPU utilization

Figure 8.6 shows the CPU utilization for the dataset server under load and at idle state. For the idle state we can see that there are spikes up to around 5% CPU utilization. When the dataset server experiences load it utilizes up to 100% CPU and depending on the workload it uses between 60% and 100% CPU.

During the idle state, the metadata extractor is sending data to the dataset server, this explains the small peaks to around 5% CPU utilization. This indicates that the amount of requests the metadata extractor sends, does not reach the capacity of the dataset server.

The load numbers shows us that the benchmarking tool can create a load heavy enough for the dataset server to use 100% CPU. But not all the commands that the benchmarking tool sends to the dataset server creates 100% CPU use. Which benchmarking commands creates the different CPU utilization is outside the scope of this experiment.

The raw experiment data is available in the files:

"benchmarks/raw_data/dataset_server_cpu_usage_data.py".

8.9 Dataset server - requests per second

Table 8.6 shows the sorted output from the Redis benchmark. The table shows us that most of the commands can service about 170,00 requests per second. These are simple requests that either push one value or gets one value from the database.

The *lrange* commands achieves less requests per second compared to the other commands. What the command does, is fetch the specified number of first items from a list. The table shows us that getting the first 600 items from a list is really expensive compared to other operations.

The measurements are the time it takes for a request to be sent from a client to the dataset server, the dataset server does some computation, and for the request to be received by the client. We also tried to use 16 pipelines for this experiment. Then the measurements reaches almost one million requests per second. These number can be seen in the raw experiment data. Pipelines combines several commands into one requests, and so this helps for commands with short computation time. Getting the first x number of items from a list does not benefit from using pipelines.

The redis-benchmark is a synthetic benchmark. It does not represent real-world scenarios. The Redis developer believes that with proper optimization the request per second numbers can be much higher than the redis-benchmark can achieve [2].

The experiment shows that the prototype does not come close to utilizing the full capacity of Redis.

The raw experiment data is available in the files:

- "benchmarks/raw_data/redis_bencmark.txt"
- "benchmarks/raw_data/redis_benchmark_pipeline.txt"

8.10 Metadata extractor - execution time

Figure 8.7 shows the execution time of the metadata extractor for three sizes of datasets. For the Small dataset the metadata extractor uses around 200 ms for each measurement. For the Medium dataset the metadata extractor uses

2. https://redis.io/topics/benchmarks

GET:	172,117.05	Requests/s
PING_BULK:	171,232.88	Requests/s
SPOP:	169,204.73	Requests/s
LPUSH:	168,067.22	Requests/s
RPOP:	167,785.23	Requests/s
SADD:	167,785.23	Requests/s
SET:	167,504.19	Requests/s
RPUSH:	167,504.19	Requests/s
LPOP:	167,504.19	Requests/s
LPUSH	166,666.66	Requests/s
PING_INLINE:	166,112.95	Requests/s
HSET:	163,934.42	Requests/s
INCR:	161,030.59	Requests/s
MSET	88,339.23	Requests/s
LRANGE_100	70,422.54	Requests/s
LRANGE_300	26,082.42	Requests/s
LRANGE_500	18,552.88	Requests/s
LRANGE_600	14,027.21	Requests/s

Table 8.6: Requests per second from Redis benchmark

around 4 seconds for each measurement. For the Big dataset the metadata extractor uses around 8 seconds.

The datasets that are used for this experiment is named Small, Medium and Big. These names are arbitrary names that are meant to reflect the ratio between the datasets. The ratio of the datasets are not accurately represented, for instance the Medium dataset uses 4% of the storage space of the Big dataset. But the Big dataset has 5% of the number of directories compared to the Medium dataset. Figure 8.7 shows that the Medium dataset uses 50% of the time the Big dataset, from this we could infer that the Medium dataset is 50% of the Big dataset, which is not the case for this experiment.

The reason why the metadata extractor uses 4 seconds on the Medium dataset and 8 seconds on the Big dataset is explained by system calls. The metadata extractor traverses the directory structure and all the files within the directory structure. The Medium dataset has 2252 directories compared to the Big dataset with 126 directories. For each directory the metadata extractor must make a system call to read from the physical hard disk. Each of the system calls are expensive.

The metadata extractor does two operations that increases the execution time. System call to the hard disk, and iteration on lists of files. Looking at fig 8.7 we

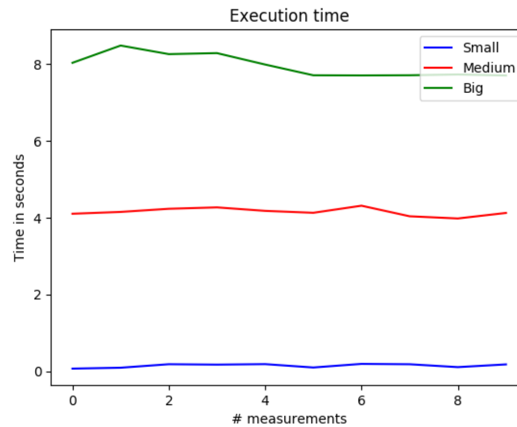


Figure 8.7: Metadata extractor execution time for Small, Medium and Big datasets

can see that the execution time is linked more to the number of system calls than the storage size of the dataset.

The raw experiment data is available in the file:

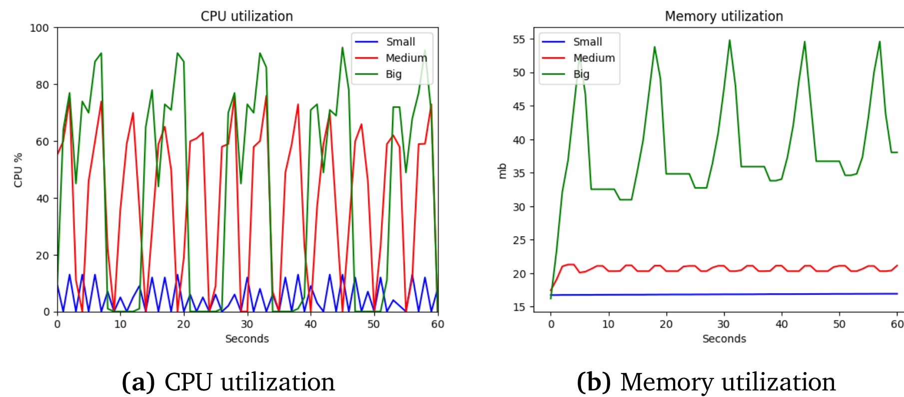
"benchmarks/raw_data/dataset_extractor_execution_time_data.py".

8.11 Metadata extractor - resource usage

Figure 8.8 shows us the resource usage for the metadata extractor. Figure 8.8a shows us the CPU utilization for the metadata extractor. The Small dataset has spikes of around 15% CPU utilization. The Medium dataset has spikes of around 75% CPU utilization. The Big dataset has spikes of around 90% CPU utilization.

Figure 8.8b shows us the memory utilization of the metadata extractor. The Small dataset uses around 16.8 mb memory for all measurements. The Medium dataset varies between 20 mb and 21 mb memory for all measurements. The Big dataset has peaks of around 50 mb and then falls to around 35 mb.

The metadata extractor runs a full iteration of gathering metadata and sending it to the dataset server. Then the metadata extractor waits for some time before the next iteration. This can be seen as the spikes in the figures. The peaks in CPU utilization is explained by the system calls and list iteration. The peaks of memory footprint is because the metadata extractor temporarily saves the metadata in memory before sending it to the dataset server.



(a) CPU utilization

(b) Memory utilization

Figure 8.8: Metadata extractor CPU utilization and memory utilization

The drop in the memory utilization for the Big dataset is explained by the Python garbage collector. The Python garbage collector is automatically freeing unused memory, and that is what we are seeing in fig. 8.8b.

This experiment shows us that the metadata extractor uses a lot of the CPU capacity and has a lot of memory that it leaves to the garbage collector. In section 9.8 we discuss how we can optimize the CPU utilization and memory footprint of the metadata extractor.

The raw experiment data is available in the file:

"benchmarks/raw_data/dataset_extractor_resource_usage_data.py".

8.12 System - CPU utilization

Figure 8.9 shows the CPU utilization of all parts of the system in an Idle state. The figure shows that Nginx and Gunicorn uses 0% CPU. Redis has spikes to 5% CPU utlization. The metadata extractor varies between 0% CPU utlization to spikes of around 90% and 40% CPU utilization.

The Redis CPU spikes are explained by Redis writing the in-memory database to disk. Nginx and Gunicorn does not use the CPU as long as they don't receive any requests.

The figure shows shows that the metadata extractor is running iterations and using between 40% and 90% CPU. Then it waits between each iteration, using 0% CPU. There are large differences in CPU utilization for the metadata

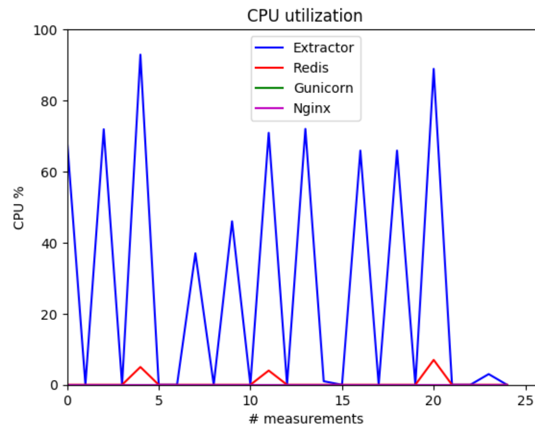


Figure 8.9: System - CPU utilization in an idle state

extractor. The reason is unknown.

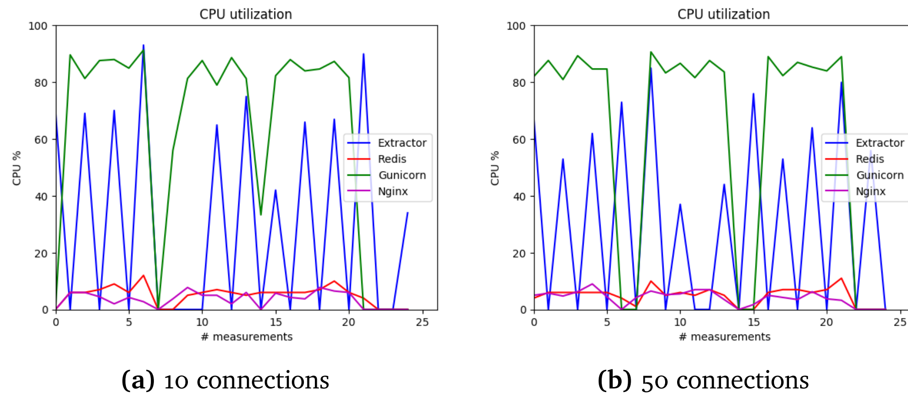
Figure 8.10 shows the CPU utilization of all parts of the system during two different workloads. The system generates about the same CPU utilization pattern for both 10 and 50 connections. The figures shows that Redis has a consistent CPU utilization of around 10%. Nginx uses between 3% and 8% CPU. Gunicorn has around 90% CPU utilization under load. The metadata extractor has the same CPU utilization pattern as in the Idle state, with spikes between 40% and 90% CPU utilization.

We can see from the figure that Gunicorn uses around 90% CPU when there are load on the system. Considering that both of the systems that interact with Gunicorn uses around 10% CPU, this points to Gunicorn being the bottleneck in this system. Gunicorn uses the default configuration, with three worker processes. There has been no effort to optimize the way Gunicorn handles requests.

The experiment shows that the two parts of the system that uses the CPU capacity is Gunicorn and the metadata extractor. If we reduce the CPU utilization of these two parts we would have free CPU capacity.

The raw experiment data is available in the files:

- "benchmarks/raw_data/system_cpu_idle_data.py".
- "benchmarks/raw_data/system_cpu_load_10connections_data.py".
- "benchmarks/raw_data/system_cpu_load_50connections_data.py".



(a) 10 connections

(b) 50 connections

Figure 8.10: System - CPU utilization under load

8.13 System - memory footprint

Figure 8.11 shows the memory footprint of all parts of the system. The memory footprint is almost identical for the system in an Idle state and during load. The figure only shows one of the workloads.

Redis starts at a few mb of memory footprint and increases to 40 mb of memory footprint. Nginx uses 17 mb memory through the whole experiment. The metadata extractor varies between 30 mb and 50 mb of memory footprint. Gunicorn consistently uses 120 mb of memory. We can see that the memory footprint for Redis increases for the first iteration of the metadata extractor. This is expected as the Redis database goes from being empty to having metadata supplied by the metadata extractor.

The experiment shows us that the memory footprint of the system is stable, and we will most likely only get an increase of the memory footprint of Redis as the database increases.

The raw experiment data is available in the files:

- "benchmarks/raw_data/system_memory_usage_data_50c.py".
- "benchmarks/raw_data/system_memory_usage_data_10c.py".
- "benchmarks/raw_data/system_memory_usage_data_idle.py".

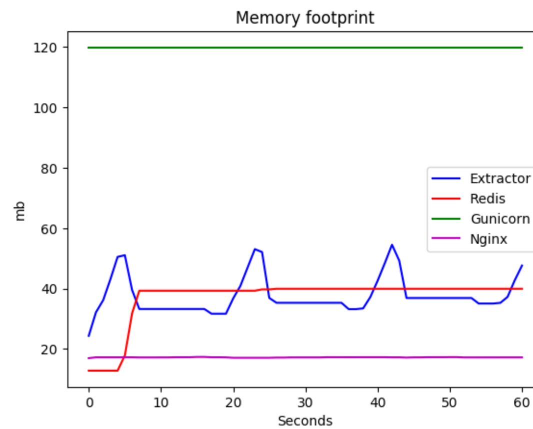


Figure 8.11: System memory utilization

8.14 os.walk

Figure 8.12 shows the execution time of os.walk in both Python 2.7 and Python 3.6. The figure shows Python 2.7 has an execution time around 1.5 seconds for each iteration. Python 3.6 has an execution time of aroun 0.5 seconds for each iteration.The result of the benchmark is that Python 3.6 uses 33% less time compare to Python 2.7.

The os.walk() function in Python 2.7 uses os.listdir() to get each entry in a directory. os.listdir() does fetch additional information for each entry but discards the additional information. For each entry it will then call os.stat() to determine if the entry is a file or directory. To get the size of a file you have to make another call to os.stat() for each entry. This gives a minimum of two system calls for each directory.

Having to make at least two system calls for each entry is expensive. That's why os.scandir() was created. [3] It removes the system call to os.stat() to determine if the entry is a file or directory.

For Python 2.7 you need to install os.scandir() as a standalone module. But from Python 3.5 and forwards the standard library os.walk() function uses the os.scandir() function directly.

The raw experiment data is available in the file:

"benchmarks/os_walk_plot.py"

3. The development of scandir: http://benhoyt.com/writings/scandir/

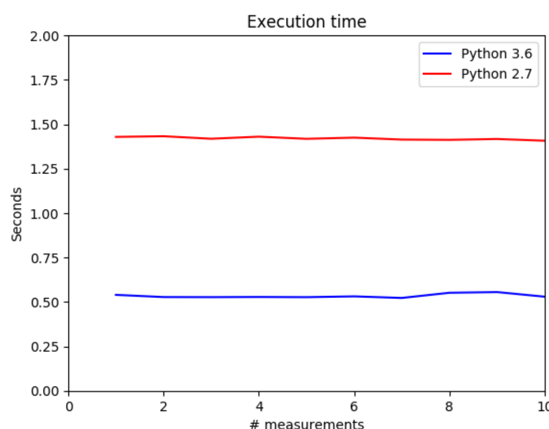


Figure 8.12: os.walk execution time in both Python 2.6 and Python 3.6

8.15 Reported disk usage

Table 8.7 shows the reported disk usage of os.walk, du -s and Nemo. From the table we can see that the experiment measures the reported disk usage of 650,608, 4139 and 1453 items. The items are both directories and files. os.walk reports 154 GB disk usage for 650,608 items. 114 mb disk usage for 4139 items. 255 mb disk usage for 1453 items.

du -s reports 152 GB disk usage for 650,608 items. 138 mb disk usage for 4139 items. 252 mb disk usage for 1452 items. Nemo reports 154 GB disk usage for 650,608 items. 114 mb disk usage for 4139 items. 255 mb disk usage for 1453 items.

Nemo reports identical disk usage as os.walk, and therefor we will not include Nemo in the following discussion. The reported disk usage for 650,608 items is 152 GB for du -s and 154 GB for os.walk. The difference is 2.31 GB or 1.5%. The reported disk usage for 4139 items is 138 mb for du -s and 114 mb for os.walk. The difference is 24 mb or 19%. The reported disk usage for 1453 items is 252 mb for du-s and 255 mb for os.walk. The difference is 2.88 mb or 1.1%.

With 650,608 items and 1453 items the difference between du -s and os.walk is around 1-2%, with os.walk reporting more disk usage. With 4139 items the difference between du -s and os.walk is 19%, with du -s reporting more disk usage. The reason for these differences may be in what the dataset includes. If there are a lot of small files, zip files and so on. This problem is not researcher in depth in this paper.

One explanation for this difference can be how a file system works, and how du

Items	650,608	4139	1453
os.walk()	154.68 GB	114.23 mb	255.83 mb
du -s	152.37 GB	138.24 mb	252.94 mb
Difference	2.31 GB	-24.01 mb	2.88 mb
Nemo	154.7 GB	114.2 mb	255.8 mb

Table 8.7: Reported disk usage

-s reports the disk usage. du -s reports the allocation space and not the absolute file space. This means that if a file is deleted but their block is not yet freed. Since du -s only is an estimate, the way the file system handles allocation and freeing of blocks are important for the estimate. The ext4 [28] filesystem uses *delayed allocation* which may have an impact on the estimate.

This experiment shows that there are differences in the reported disk usage for different utilities.

9

Discussion

9.1 Thesis

The idea includes the abstract concept of Information. Chapter 3 defines the concept of Information as organized and processed metadata. A researcher wants to analyze metadata about a dataset. The system organizes and processes the metadata before creating a visualization. We wanted to create an abstraction of the concept of organizing and processing metadata. We choose to call the abstract concept: Information. We assume that the reader understands when we are refering to information (generalized) and Information as an abstract concept.

The architecture for the project isolates each part of the prototype into systems. The Instrument extracts metadata from any dataset and deliver metadata to any client. The Visualization abstraction in this prototype is dependent on specific types of metadata to be able to create visualizations.

The design of the system realizes the architecture into one out of several possible systems. For example the web server is just the functionality of requesting metadata from the dataset service and delivering the metadata as information to the Visualization.

The implementation of the prototype uses Python for every part of the system. This allowed us to quickly create working solutions to each part of the system. Python is a interpreted language [29] and is executed step-by-step at runtime.

Compared to a compiled language [30] that compiles to machine code and in the compile process can optimize code execution. This means that in many cases Python perform slower than a compiled language [31][32][33].

The experiment shows that we implemented a system that can execute together. We observe around 90% CPU utilization for the metadata extractor and Gunicorn. The other parts have below 20% CPU utilization. The experiment show that Redis has a predictable memory footprint that allows for calculations of the future memory footprint. The other parts of the system has a stable memory footprint with Gunicorn having the biggest footprint of 120 mb. Giving us free memory capacity that Redis can utilize.

9.2 Optimizing Redis

For this prototype we store most of the metadata for a directory in a Redis hash. We did this because it is logical to have a hash that represents a directory, and all the associated metadata contained within the hash. The COAT dataset structure has directories that are structured first by year, then location and date. This gives us possibilities to "tag" information and sort them into sets. Redis sets support unions, intersections and differences. A set is a collection of unique strings. For example we could create a set for each location per year. The set includes all the dates when the location has had measurements. This would allow us to create complex queries such as "has this location had any measurements on these specific dates the last few years?".

We can optimize the way we store metadata collected over time. We can use one sorted set for each type of metadata. The score is the measurement time and the value is the metadata. This would give us a way to request metadata for a directory over time, such as the number of files in a directory over time. We have the ability to get that metadata now, but it involves as many requests as there are measurements for each directory.

The "/overview" page creates a new request for each directory just to get the modified date. If we store the modified date in one sorted set, this would enable the /overview page to use one requests to get the modified date for all directories. Storing metadata over time in sorted sets enables the system to use Redis commands for sorted sets such as requesting a specified range of metadata.

Another solution for the /overview page is to reduce the number of requests it sends. Utilizing Redis pipelines (see section 6.3.3) we could bundle together several requests. If the web server bundles together every 100th request in a

pipeline, the experiment results from table 8.2 would reduce Firefox network requests from about 260 requests to around 10 requests.

The "history" command that the web server offers can service around 800 requests per second compare to the other commands that can service 2500-3000 requests per second, as seen in figure 8.2b. The command gets a list of historic metadata entries from the dataset server. For each entry in the list, the web server must request the metadata from the dataset server. We can reduce the "history" cost by utilizing both sorted sets and pipelines.

9.3 Scale

How we scale the capability of a computer system has been researched for a long time and there are several solutions to the problem. We can scale vertically, horizontally, distribute, replicate and so on. This is not the case with visualization. Most visualizations [9] show one metric, such as value or occurrences. For one or more items such as people, companies or different rates (unemployment, obesity, army movement). The visualization may also include the relation of the items such as time, hierarchy or geographical relation.

The human user must understand and be able to navigate the visualization. If the user understands the visualization, the user can create conclusions based on the understanding. When we create the prototype, do we focus our work on a users first time experience or do we assume that the user has prior knowledge of the system. Creating a new way of visualizing information involves learning how to interpret the visualization. Most human users know how to interpret visualization techniques that are used today.

There is a visual information-seeking mantra[13]:

> Overview first, zoom and filter, then details-on-demand.

Creating an overview can be difficult if there are many items that we want to visualize. We can reduce the number of items by filtering the information that the visualization application uses to create the visualization. But if the visualization application filters the information before the user can see it, how can the visualization application determine if it filtered the correct information? The human user still wants to analyze all of the information. And the user may need an overview of the information to understand what to filter out.

There are examples of big data visualization that accomplish big scale visualization. The examples are space visualizations [34] [35] [36]. They have one

	1 hour	1 year
Average size	44 b	196 mb
Worst size	4175 b	18.6 GB

Table 9.1: Calculated memory utilization

thing in common, the sun. The visualizations uses the sun as an anchor point, and it helps the human user to navigate the visualizations.

9.4 The amount of keys

9.4.1 One year of measurements

If we use the numbers from the "Dataset server - keys" experiment in section 8.6 we can calculate the memory usage for a year of measurements. One year is 8760 hours, and if we do a measurement once every hour it gives us 8760 measurements for a year. The worst case scenario is that each time we do a measurement we create all new keys, which is not the case in the prototype. Running the metadata extractor once using the Big dataset creates 509 keys, the keys are both sets, hashes, lists and sorted sets.

The worst case scenario would create 509 new keys every hour for 8760 measurements, giving a total of 4,458,840 keys. If we use the average size of all keys from table 8.5 and the worst average size from table 8.4 we can create the table 9.1. Table 9.1 shows that if take the average size of all types of keys and creates over four million keys we get a total size of 196 mb. If we take the worst average size of a set and create over four million keys we get a total size of 18.6 GB.

Since there is such a big difference between the worst size and the average size, we can look closer at a specific data types. One measurement of the Big dataset creates 252 hashes and for each measurement we create a new hash. 252 hashes 8760 times, gives us a total of 2,207,520 hashes for a year. Each hash has an average size of 10 bytes, which gives us a total size of 22 mb.

Looking at the theoretical numbers that is presented here, we can see that the prototype is well suited to save metadata. The average size of 44 bytes gives a worst case scenario of 196 mb for a year of metadata. The worst size calculation would mean that we saved all filenames 8760 times. The prototype visualization doesn't actually use the filename sets, and we could remove it from the prototype. This would give us an lower calculated size.

Data type	# types	Average size	1000 iter.	9000 iter.
list	126	1000 b	126 kb	1.1 mb
set	123	4175 b	513 kb	0.5 mb
hash	126126	10 b	1261 kb	11.3 mb
zset	8	785 b	6 kb	0.05 mb
Total			1907 kb	13 mb

Table 9.2: Theoretical memory footprint of Big dataset

9.4.2 Dataset growth

In order to better understand how the Big dataset would evolve over time, we let the metadata extractor run for 1000 measurements. Note that the dataset was static for every measurement, meaning that there are no new files. After 1000 measurements there are a total of 126,383 keys with a total size of 5.6 mb. Based on the measurements we can do some calculations to find the memory footprint after one year and around 9000 measurements.

Table 9.2 shows the number of each data type after 1000 iterations and the average size of each data type. Comparing these numbers with the experiment measurements in table 8.4, we can see that the list has the same number of items, but the average size has increased. Meaning that there are the same number of lists, but each list has more elements. Zset (sorted set) also has the same number of items with an increase of average size, meaning that each sorted set has more elements. The sets have not changed, they have the same number of items, each item has the same number of elements and the average size is the same. This is because this prototype only stores filenames in the sets, and there are no change in the amount of filenames. For hashes there are 126,126 items after 1000 iterations compared to 3150 hashes in table 8.4. The increase is because we create new hashes for each directory for each measurement. The average size is the same in both instances.

This gives us a calculated size of 13 mb for one year of metadata measurements. It will take over 690 years to reach 9 GB of memory footprint for Redis. This means that we have free capacity to utilize Redis data structures as discussed in section 9.2.

The raw experiment data for 1000 measurements is available in the file:

"benchmarks/rar_data/theory/Big_dataset_1000_iterations".

9.5 Prototype bottleneck

Table 8.6 shows the requests per second that Redis could service. Redis serviced around 170,000 requests per second for the experiment and the worst case for the experiment was 14,000 requests per second. Figure 8.1 shows the requests per second that the web server could serve through Nginx. The web server served around 2500 requests per second. Figure 8.2 shows the requests per second that Gunicorn could service. Gunicorn serviced around 2500 requests per second. Figure 8.10 shows the CPU utilization of all parts of the system. Gunicorn uses around 90% CPU compared to Redis and Nginx that uses around 10%.

Based on these numbers we can conclude that Gunicorn is the bottleneck in the system. Removing Nginx from the requests chain did not change the requests per second measurement. And sending requests directly to Redis shows a much higher worst case result. From the CPU utilization experiment we can see that Gunicorn uses most of the CPU capacity available.

9.6 React, abstractions and Google charts

React is based on encapsulated components and the principle of creating abstraction layers. When we are in a learning phase, creating abstractions for something you don't understand is difficult. When the abstraction is made by someone else, this makes it even harder to learn. This means you have to learn the underlying system and how the abstraction works.

The Google Charts library is a library with support for 18 native chart types, 18 additional chart types and five types of controls. Google Charts has an abstraction layer that allows the user to bundle together several types of charts and controls.

React is set up in a way that encourages the use of components. Components are abstractions that are meant to make it easier for developer to reuse parts of the application. For example, if you have a main view for the application, the main view is split into components. An example main view is <Header><Text><Table><Chart>. Each of these parts points to a component. Each of the components can include new views (which includes new or reused components). Looking at the <Chart> component from the example main view, this component can consist of <Text><LineChart><Text>. The component <LineChart> is the component that "directly" uses the react-google-charts component. Which itself is an abstraction layer.

The original Google Charts library is developed by Google. react-google-charts is developed by one independent developer. At the time of using the react-google-charts component, all of the Google Charts functionality was not supported. This gave me the option to either write support for additional functionality or use the Google Charts library directly. I ended up with using the Google Charts library directly.

9.7 Extract metadata on dataset change

The prototype runs the metadata extractor as a process that starts a new iteration on a set interval. Another solution that was considered was having the metadata extractor execute when there is a change in the filesystem. This solution involves monitoring every directory that we want to extract metadata from for changes. Linux has a system, Inotify [37], that can monitor the filesystem and notify if there are changes to the filesystem. The reason we can not use Inotify for this system is that Inotfy does not support recursively watching directories. Meaning that a separate Inotify watch must be created for every subdirectory.

Inotify has a default configuration "max user watches" limit of 8192 files you can monitor. And each watcher for Inotify opens a file descriptor, and therefore Inotify is limited by the max number of files the file system allows a user to have. That limit is defined by *ulimit* which is a Linux system call that limits the use of system-wide resources. For example on the development machine the default maximum number of open files are 1024. The ulimit default configuration can be changed, but for this prototype we did not choose to do it.

9.8 Metadata extractor resource usage

Figure 8.8 shows the CPU utilization and memory utilization of the metadata extractor. The Small dataset has a unrealistic size, so for this discussion we will ignore the Small dataset. Figure 8.8a shows a CPU utilization of 60-90% for both the Big and Medium dataset. The metadata extractor uses a sequential extraction process and is a single-threaded process. The CPU utilization measurements in fig. 8.8a are for one CPU core. If we could split the workload to several threads, we could utilize the CPU cores and reduce the peaks of CPU utilization. For example use 20% CPU on four cores rather than 80% on one core.

If we define that the metadata extractor will collect metadata once every hour,

we could increase the execution time of the metadata extractor. This can be achieved either by having the extractor wait at certain points, or by setting a limit on the CPU utilization. Python has a resource module [38] that can set a maximum amount of processor time that a target process can use.

Figure 8.8b shows a memory footprint of 20 mb for the Medium dataset and up to 50 mb for the Big dataset. The memory footprint of the metadata extractor is not problematic, but we could reduce the memory footprint by sending the collected data as it is collected. The current prototype collects and stores all metadata before sending it all in one operation. Sending metadata when it is collected would reduce the memory footprint, but it would create more requests for the dataset server.

10

Contributions

We contribute an architecture and design of a system that enables human users to interact with and visualize information. The visualization can request new information and change how the visualization is created based on input from the human user.

We contribute an architecture and design of a system that uses in-memory storage to store information about the history of a dataset. The system extracts metadata about a dataset and stores it in-memory. The system enables access to the stored information.

We implement an artifact prototype system that validates the design of the system. The implementation utilizes Redis as an in-memory data structure store. The data structures enables the information to be logically mapped and stored.

We evaluate the system and observe that we can reduce the CPU utilization of the metadata extractor and Gunicorn. The evaluation show that Redis has a predicable memory footprint.

We contribute a validation of the idea of using in-memory storage to store dataset information over time. Redis allows us to use data structures that enables logically mapping information. Our calculations indicate that Redis can store over 690 years of metadata with a memory footprint of 9 GB.

11 Summary and Conclusion

The goal of this Master's thesis is to enable researchers to visualize information about a dataset with the purpose of looking for trends and identify changes over time. The information is organized and processed metadata about a dataset that is collected over time. This thesis describes how we design, implemented and test a solution that achieves the goal.

We have implemented a metadata extractor in Python that can extract metadata from a dataset and send it to a dataset sever that stores the metadata. The dataset server is realized utilizing Redis as an in-memory database. We have implemented a web server using Django, Gunicorn and Nginx. The web server can request metadata from the dataset server and transform metadata into information that is supplied to the visualization application. The visualization application is a JavaScript web application that creates visualizations based on the information supplied by the web server. The researchers uses the visualization application to access, monitor and analyze historic information about the dataset.

Our experiments shows us that the web server can serve about 2500 web pages to 10 concurrent connections with a latency below 5 ms. And the web server can server about 2500 web pages to 50 concurrent connections with a latency below 30 ms. This would stress the Gunicorn web server and it would use around 90% CPU. The experiments show that Redis has free capacity both in regards to the number of requests it can serve, but also the CPU utilization. The memory footprint of Redis is dependent on the amount of information

that we store. Our experiments show that the hash data type, which is the main method that we store metadata, has an average size of 10 bytes. One experiment shows that storing 100 million key-value pairs in Redis uses 9 GB of memory. The key-value pairs has an average size of 8.5 bytes.

The thesis contributes an architecture and design that enables and supports the creation of visualizations of organized and processed metadata. The artifact validates using in-memory storage to store the historic metadata.

12 Future work

In section 9.5 we have identified the Gunicorn web server as a bottleneck for information requests. We could try to change from a Python based web server to another technology such as Golang [1]. The web server does not need to support complex routing logic or support database migrations, and Golang offers a simple and effective HTTP package.

We have identified that the metadata extractor utilizes most of the CPU capacity of the CPU core it is using. We have detailed that we could try to parallelize the workload and limit the CPU utilization of the prototype. The metadata extractor defines the minimum time between each measurement of the dataset, as the execution of the metadata extractor must complete before a new measurement can begin. If the time between each measurement is longer than the execution time of the metadata extractor, this means that the resource utilization can be lowered and have a longer execution time.

In this prototype the metadata extractor is a process that a human user must manually execute and monitor. If the metadata extractor crashes a human user must notice it and restart the process. The metadata extractor could be a service that the OS executes and monitor. The metadata extractor could have a interface for users to input a different measurement interval.

The communication with Redis from the metadata extractor uses pipelines, this

1. https://golang.org/

optimizes the communication between them. The communication between the web server and Redis does not use pipelines, and for some of the information requests pipelines can drastically improve the communication overhead. The information extraction from Redis could also be improved if we improve the way we store information in Redis. By utilizing sorted sets for more information, we can improve the number of requests needed for certain types of information. This would also enable the system to create complex queries that involves several sets of information.

The visualization application shows that it is possible to visualize the information, and gives examples of visualization techniques that is appropriate. There is a lot of possibility for future work with the visualization application. For each web page we could add more controls both for controlling the visualization but also changing the way the visualizations are created. Currently there are no styling for the visualization application.

The current system implements one dataset server, the design of the system is to have multiple dataset servers. We could implement support for more dataset servers in the web server and add options to query the additional dataset servers in the visualization application. The whole system would run on each machine, and the web server and visualization application is aware of the other dataset servers.

The Road Towards the Artifact

This chapter details the approach used and issues seen while progressing towards the described prototype. The idea of our approach was to create a system that allowed us to iteratively try different visualization approaches. The system has three components; Visualizer, Instrument and Dataset.

Visualizer The visualizer displays visual information to the user. The visualizer will not use common visualization techniques such as charts and tables. The visualizer uses the information that is collected by the instrument to create a visualization.

Instrument The instrument extracts information from the dataset. This includes information about the dataset and information about each element in the dataset. The instrument stores the extracted information.

Dataset The dataset that we use for our approach is a file system. A file system includes directories and files. The dataset includes metadata that the file system has stored about the files and directories. The metadata include:

- Size of file system

Name	Size	Quota
Directory1	600	40%
Directory2	200	13%
Directory3	700	47%
Total	1500	100%

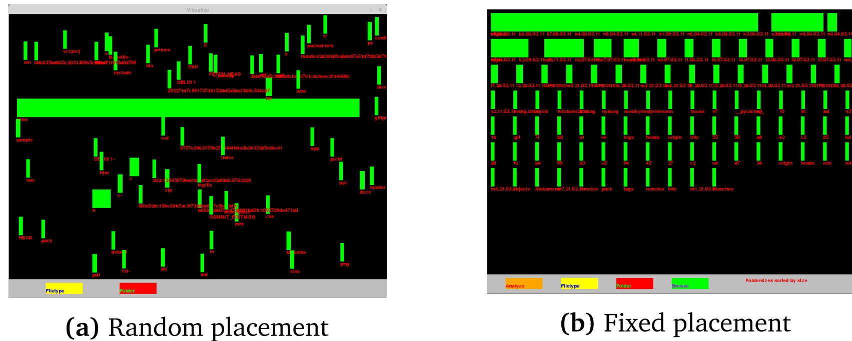
Table A.1: Directory size quota

- Name of each directory
- Name of each file
- Last modified time
- File type
- Total number of directories
- Total number of files
- Size of a directory
- Size of a file

A.1 Approach 1

For approach 1 we focused on visualizing metadata as a visual object. We arbitrarily chose a square shape. The square represents one piece of metadata. The size of the square is based on the value of the metadata. If we directly translate the value of the metadata to the size of the square, we can have squares that are larger than the screen area. The solution we used for this problem was to create a value quota. For example the directory size can be 300,000 kb, which can't be directly assigned as the size of a visual object on screen. Table A.1 shows how we created a quota of the directory size. All directories add up to 100%, and we can use the percentage as the on screen size of a directory.

Figure A.1a shows our first attempt to place the square shapes on the screen. The squares are placed randomly on the screen. This is a simple placement algorithm that only needs to check if the square is inside the screen area and does not overlap any other square. In our opinion this does not aid us to



(a) Random placement

(b) Fixed placement

Figure A.1: Version 1 square placement

visually compare items. We determined that comparing items is important for the visualization.

Our second attempt to place the squares on the screen can be seen in fig. A.1b. The squares are organized and sorted on the screen. The placement algorithm precisely calculates the placement of each square. We found that it's important to have padding around each square. The squares are sorted by value, and in our opinion this approach allows us to compare items. But the figure shows us that several squares have the same size even though the value is different. This is because the quota of the squares are skewed towards the biggest squares. In fig. A.1b the biggest squares has a quota of around 90% and the smallest squares are under 1%. This means that we cannot visually differentiate the smallest squares even if their value are different.

A.2 Approach 2

For approach 2 we focused on sorting the square visual objects. To sort the objects we determine that we want an overview of all objects. The overview has each square representing the value of the metadata as a quota, similar to approach 1. In addition to the overview we want to sort all the objects into smaller groups. The groups are sorted with the biggest squares in one group, and the smaller squares in their own group. Each square inside the groups have the same visual size, but they are sorted based on the value. The size difference between squares inside a group is visualized by having the biggest items blink. How much and how quickly they blink is based on the value. The biggest squares blink faster. To identify the groups in the overview, we use a unique color for each group and use that color for those squares in the overview.

Figure A.2 shows our first attempt to create an overview and place the squares

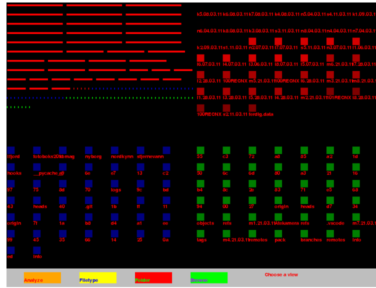
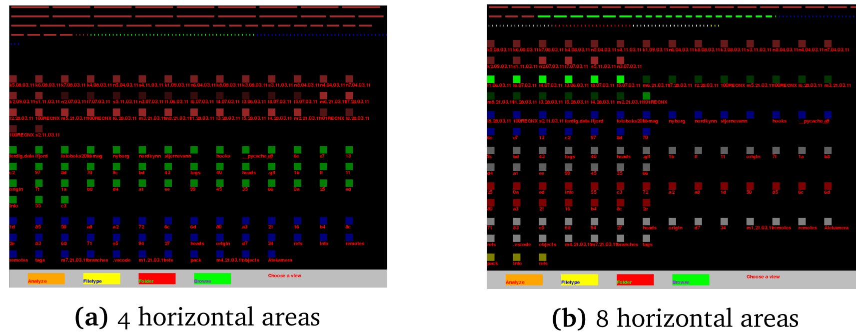


Figure A.2: Version 2 static placement



(a) 4 horizontal areas **(b)** 8 horizontal areas

Figure A.3: Version 2 dynamic placement

in groups. The placement of the overview and the groups needs to calculate both the vertical and horizontal placement. Having to consider both the vertical and horizontal placement made the placement algorithm complex. A problem with this approach is that it's not immediately obvious how a user should read the visual information.

Our second attempt is seen in fig. A.3. Placing the overview and groups vertically means that each group fills the whole width, and we only need to consider the horizontal placement. It makes the placement algorithm less complex compared to our first attempt. Having a less complex placement algorithm allows us to give the user the ability to change the number of groups to sort the squares into. Figure A.3a shows the squares sorted into three groups. Figure A.3b shows the squares sorted into seven groups. We feel that having more groups allows us to identify smaller size differences between the squares. But at the same time we feel it makes the visualization cluttered.

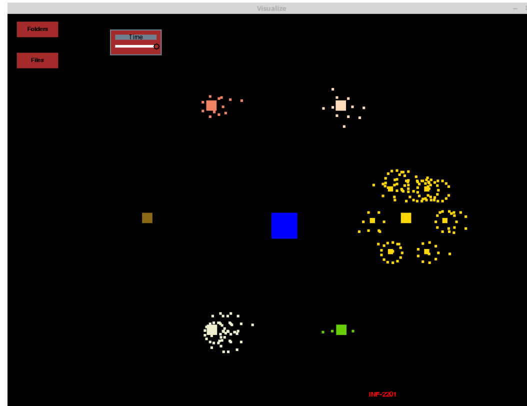


Figure A.4: Circular placement

A.3 Approach 3

For our third approach we focused on visualizing how directories are connected. The visualization will recreate the directory hierarchy based on a circle placement algorithm, seen in fig. A.4. The main root of the hierarchy is placed in the middle of the screen. All subdirectories of the root is placed on the circumference of a circle around the root. Each subdirectory of the current root is the root of a new circle pattern. The main root has a unique color. Each subdirectory and it's subdirectories are given a unique color.

Storing information about how directories are connected allows the user to click on a subdirectory to focus on that directory. This makes the selected directory the new main root of the visualization. This functionality allows the user to navigate the directory hierarchy.

We created an option for the user to filter which directories are visualized. The filter is based on the last modified time metadata. This allows the user to only visualize the directories that has a last modified time before a certain time.

In our opinion this approach achieves most of what it wanted to achieve. We manage to visualize how directories are connected. We allow the user to use the visualization to navigate the directory hierarchy. But this approach also highlights the problems that are associated with visualizations. If there are more than a hundred directories, we feel that the visualization looses it's clarity. And it makes it hard to identify and differentiate the different directories. There is also a problem visualizing more than 4-5 layers of subdirectories.

A.4 Conclusion

The idea of this research was to give a human user the ability to view visual information about a data volume. A part of the idea was to not use any conventional visualization techniques such as charts, diagrams and tables. During the iterations of this research we tried using squares that represent information in different ways. Either as a representation of the underlying information or as a part of a bigger picture. We also tried using a circular approach, with circles representing information, and each circle having a circle of related information. The technical creation of the visualizations was successful, but we learned something about the creation and practical use of a new type of visualization.

We tried several visualization approaches both fancy and basic to see what was needed from the Instrument. The Instrument is the system that enables other systems to gather and visualize information. It is important that the Instrument supports flexibility. The problem statement is therefore not how do we visualize the information, but how do we support the creation of the visualization.

/B

Redis key size

```
human_size() {
        awk -v sum="$1"
        ' BEGIN {hum[1024^3]="Gb";
        hum[1024^2]="Mb"; hum[1024]="Kb";
        for (x=1024^3; x>=1024; x/=1024){
                if (sum>=x) {
                        printf "%.2f %s\n",sum/x,hum[x];
                        break;
                }
        }
        if (sum<1024) print "1kb"; } '
}

redis_cmd='redis-cli '

# get keys and sizes
for k in `$redis_cmd keys "*"`;
do key_size_bytes=`$redis_cmd
debug object $k |
perl -wpe 's/^.+serializedlength:([\d]+).+$/$1/g'`;
size_key_list="$size_key_list$key_size_bytes $k\n";
done

# sort the list
```

```
sorted_key_list=`echo -e "$size_key_list" | sort -n`

# print out the list with human readable sizes
echo -e "$sorted_key_list" | while read l; do
    if [[ -n "$l" ]]; then
        size=`echo $l | perl -wpe 's/^(\d+).+/$1/g'`;
        hsize=`human_size "$size"`;
        key=`echo $l | perl -wpe 's/^\d+(.+)/$1/g'`;
        printf "%-10s%s\n" "$hsize" "$key";
    fi
done
```

Listing B.1: https://gist.github.com/epicserve/5699837

C
Redis mass-insertion

```
import sys

def proto(line):
    result = "*%s\r\n$%s\r\n%s\r\n" % (str(len(line)),
                    str(len(line[0])), line[0])
    for arg in line[1:]:
        result += "$%s\r\n%s\r\n" % (str(len(arg)), arg)
    return result

if __name__ == "__main__":
    try:
        filename = sys.argv[1]
        f = open(filename, 'r')
    except IndexError:
        f = sys.stdin.readlines()

    for line in f:
        print proto(line.rstrip().split(' ')),
```

Listing C.1: https://github.com/TimSimmons/redis-mass-insertion

Bibliography

[1] A. Inc., "How to check the storage on your iPhone, iPad, and iPod touch." https://support.apple.com/en-us/HT201656. [Online; accessed 09-May-2018].

[2] W. contributors, "Disk Usage Analyzer — Wikipedia, The Free Encyclopedia." https://en.wikipedia.org/w/index.php?title=Disk_Usage_Analyzer&oldid=839026927. [Online; accessed 09-May-2018].

[3] J. Liu, T. Tang, W. Wang, B. Xu, X. Kong, and F. Xia, "A Survey of Scholarly Data Visualization." In: 1IEEE Access, vol. 6, (2018), pp. 19205-19221. DOI: 10.1109/ACCESS.2018.2815030 URL: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=8314667&isnumber=8274985. [Online; accessed 12-May-2018].

[4] N. T. X. Qin, Y. Luo and G. Li, "DeepEye: An automatic big data visualization framework." In: Big Data Mining and Analytics, vol. 1, no. 1, (2018), pp. 75-82. DOI: 10.26599/BDMA.2018.9020007 URL: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=8268737&isnumber=8268729 . [Online; accessed 12-May-2018].

[5] A. Figueiras, "Towards the Understanding of Interaction in Information Visualization." In: 19th International Conference on Information Visualisation, Barcelona, (2015), pp. 140-147. DOI: 10.1109/iV.2015.34 URL: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=7272592&isnumber=7272518. [Online; accessed 12-May-2018].

[6] A. Holupirek and M. H. Scholl, "Implementing filesystems by tree-aware DBMSs." In: Implementing filesystems by tree-aware DBMSs. Proc. VLDB Endow. 1, 2, (2008), pp. 1623-1630. DOI: http://dx.doi.org/10.14778/1454159.1454237 URL: https://dl.acm.org/citation.cfm?id=1454237. [Online; accessed 25-April-2018].

[7] G. Hackl, W. Pausch, S. Schönherr, G. Specht, and G. Thiel, "Synchronous metadata management of large storage systems." In: Synchronous metadata management of large storage systems. In Proceedings of the Fourteenth International Database Engineering & Applications Symposium (IDEAS '10). pp. 1-6. DOI: 10.1145/1866480.1866481 http://doi.acm.org/10.1145/1866480.1866481 URL: `https://dl.acm.org/citation.cfm?id=1866481`. [Online; accessed 25-April-2018].

[8] P. Mundkur, V. Tuulos, and J. Flatow, "Disco: a computing platform for large-scale data analytics." In: Disco: a computing platform for large-scale data analytics. In Proceedings of the 10th ACM SIGPLAN workshop on Erlang (Erlang '11). pp. 84-89. DOI: https://doi.org/10.1145/2034654.2034670 URL: `https://dl.acm.org/citation.cfm?id=2034670`. [Online; accessed 25-April-2018].

[9] J. Heer, M. Bostock, and V. Ogievetsky, "A tour through the visualization zoo." In: A tour through the visualization zoo. Commun. ACM 53, 6, (2010), pp. 59-67. DOI: https://doi.org/10.1145/1743546.1743567 URL: `https://dl.acm.org/citation.cfm?id=1743567`. [Online; accessed 06-April-2018].

[10] M. Kyritsis, S. R. Gulliver, S. Morar, and R. Stevens, "Issues and benefits of using 3D interfaces: visual and verbal tasks." In: Issues and benefits of using 3D interfaces: visual and verbal tasks. In Proceedings of the Fifth International Conference on Management of Emergent Digital EcoSystems (MEDES '13), (2013), pp. 241-245. DOI: http://dx.doi.org/10.1145/2536146.2536166 URL: `https://dl.acm.org/citation.cfm?id=2536166`. [Online; accessed 06-April-2018].

[11] K. Z. Quang Vinh Nguyen, Mao Lin Huang and I.-L. Yen, "A Visualization Model for Web Sitemaps." In: International Conference on Computer Graphics, Imaging and Visualisation (CGIV'06), Sydney, Qld., (2006), pp. 12-17. DOI: 10.1109/CGIV.2006.14 URL: `http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=1663761&isnumber=34835`. [Online; accessed 05-April-2018].

[12] W. Wang, H. Wang, G. Dai, and H. Wang, "Visualization of Large Hierarchical Data by Circle Packing." In: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, (2006), pp. 517-520. DOI: 10.1145/1124772.1124851 URL: `http://doi.acm.org/10.1145/1124772.1124851`. [Online; accessed 05-April-2018].

[13] B. Shneiderman, "The eyes have it: a task by data type taxonomy for information visualizations." In: Proceedings 1996 IEEE

Symposium on Visual Languages, Boulder, CO, (1996), pp. 336-343. DOI: 10.1109/VL.1996.545307 URL: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=545307&isnumber=11360. [Online; accessed 05-April-2018].

[14] W. contributors, "Web Server Gateway Interface — Wikipedia, The Free Encyclopedia." https://en.wikipedia.org/w/index.php?title=Web_Server_Gateway_Interface&oldid=837620822. [Online; accessed 30-April-2018].

[15] B. Chesneau, "Deploying Gunicorn – Gunicorn 19.8.0 documentation." http://docs.gunicorn.org/en/latest/deploy.html. [Online; accessed 30-April-2018].

[16] S. Sanfilippo, "An introduction to Redis data types and abstractions." https://redis.io/topics/data-types-intro. [Online; accessed 30-April-2018].

[17] S. Sanfilippo, "Using pipelining to speedup Redis queries." https://redis.io/topics/pipelining. [Online; accessed 04-May-2018].

[18] G. Rodola, "psutil documentation." http://psutil.readthedocs.io/en/latest/index.html?highlight=memory%20info#psutil.Process.memory_info. [Online; accessed 01-May-2018].

[19] S. Sanfilippo, "How fast is Redis?." https://redis.io/topics/benchmarks. [Online; accessed 02-May-2018].

[20] Google, "Chrome DevTools Overview." https://developer.chrome.com/devtools. [Online; accessed 01-May-2018].

[21] Mozilla, "Firefox Developer Tools." https://developer.mozilla.org/son/docs/Tools. [Online; accessed 01-May-2018].

[22] S. Sanfilippo, "Redis Mass Insertion." https://redis.io/topics/mass-insert. [Online; accessed 01-May-2018].

[23] T. L. I. Project, "The du Command." http://www.linfo.org/du.html. [Online; accessed 12-May-2018].

[24] W. contributors, "Nemo (file manager) — Wikipedia, The Free Encyclopedia." https://en.wikipedia.org/w/index.php?title=Nemo_(file_manager)&oldid=834287030. [Online; accessed 12-May-2018].

[25] Restuta, "Sizes of JS frameworks." https://gist.github.com/Restuta/cda69e50a853aa64912d. [Online; accessed 02-May-2018].

[26] W. contributors, "Minification (programming) — Wikipedia, The Free Encyclopedia." https://en.wikipedia.org/w/index.php?title=Minification_(programming)&oldid=836196003. [Online; accessed 02-May-2018].

[27] R. Nelson, "Tuning NGINX for Performance - NGINX." https://www.nginx.com/blog/tuning-nginx/. [Online; accessed 08-May-2018].

[28] W. contributors, "Ext4 — Wikipedia, The Free Encyclopedia." https://en.wikipedia.org/w/index.php?title=Ext4&oldid=835235234. [Online; accessed 12-May-2018].

[29] W. contributors, "Interpreted language — Wikipedia, The Free Encyclopedia." https://en.wikipedia.org/w/index.php?title=Interpreted_language&oldid=839980144. [Online; accessed 11-May-2018].

[30] W. contributors, "Compiled language — Wikipedia, The Free Encyclopedia." https://en.wikipedia.org/w/index.php?title=Compiled_language&oldid=836186637. [Online; accessed 11-May-2018].

[31] I. Gouy, "Python 3 programs versus Go." https://benchmarksgame-team.pages.debian.net/benchmarksgame/faster/python3-go.html. [Online; accessed 11-May-2018].

[32] kostya, "Some benchmarks of different langugages." https://github.com/kostya/benchmarks. [Online; accessed 11-May-2018].

[33] P. S. Foundation, "Why is Python slower than the xxx language." https://wiki.python.org/moin/Why%20is%20Python%20slower%20than%20the%20xxx%20language. [Online; accessed 11-May-2018].

[34] I. Webster, "3D Interactive Asteroid Space Visualization." http://www.asterank.com/3d/. [Online; accessed 04-May-2018].

[35] G. D. A. Team, "100,000 stars." http://stars.chromeexperiments.com/. [Online; accessed 04-May-2018].

[36] INOVE, "Solar System Scope - Online Model of Solar System and Night Sky." https://www.solarsystemscope.com/. [Online; accessed 04-May-2018].

[37] W. contributors, "Inotify — Wikipedia, The Free Encyclopedia." https://en.wikipedia.org/w/index.php?title=Inotify&oldid=838189851. [Online; accessed 08-May-2018].

[38] P. S. Foundation, "resource – Resource usage information." https://docs.python.org/3/library/resource.html. [Online; accessed 10-May-2018].