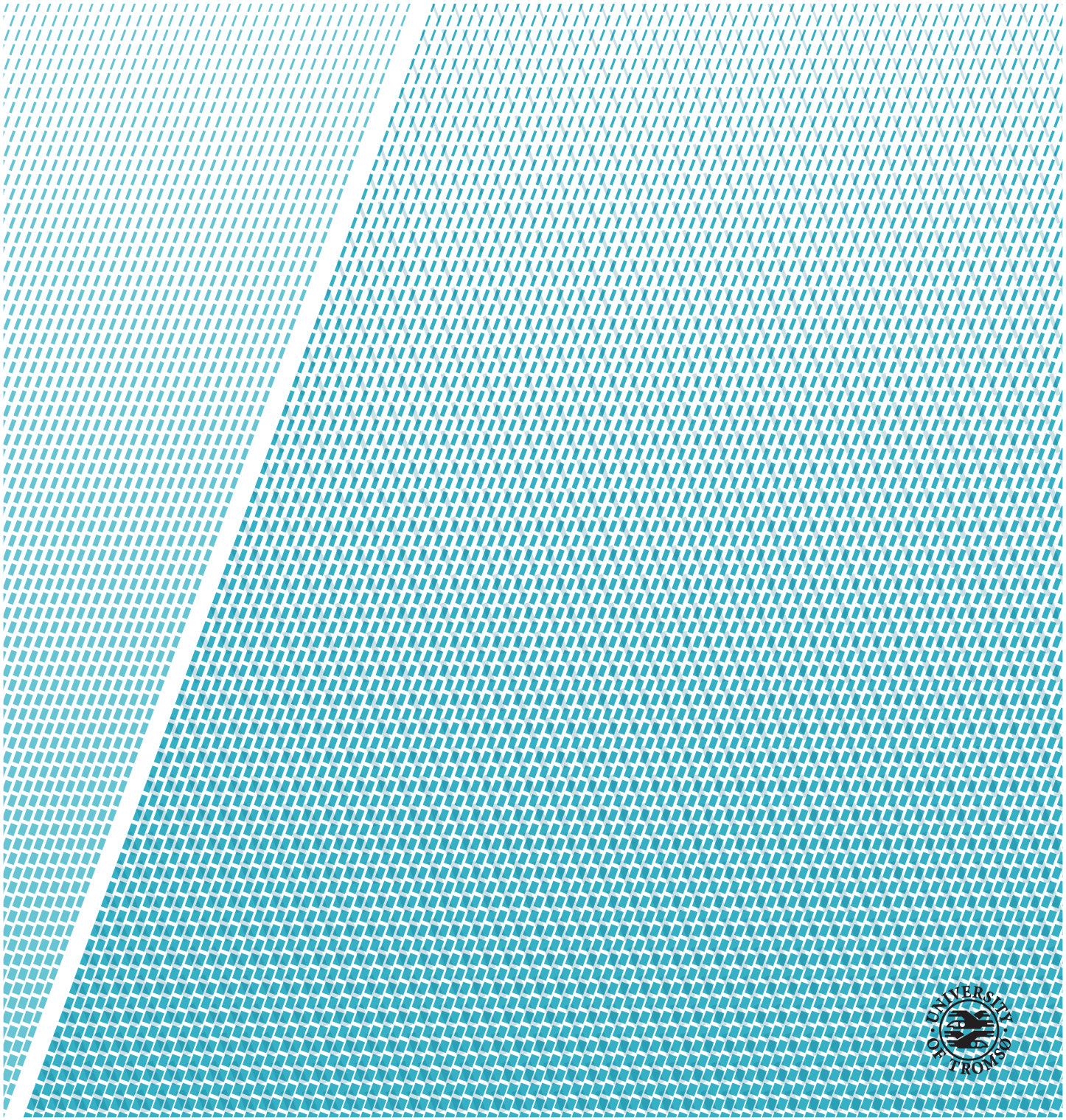
UiT
THE ARCTIC
UNIVERSITY
OF NORWAY

Faculty of Science and Technology
Department of Computer Science

From Physical to Virtual Sensors (PVS)

Camilla Stormoen

INF-3983 Capstone Project in Computer Science . . . December 2017



Abstract

The Arctic Tundra in the far northern hemisphere is one of ecosystems that are most affected by the climate changes in the world today. Every year, COAT deploys several camera traps in eastern Finnmark, Norway during the winter to study predator populations. These cameras are collecting high volume of images that results in Big Data challenges in the ecology field. Managing the sensors and collecting data is currently a manual task that can introduce many errors and there is a need for systems and abstractions which makes it simple to build, maintain and use sensors and sensor data.

This project describes a system that simplifies biologists need to look at images from different locations in the Arctic Tundra. The purpose is to provide for a more powerful and flexible sensor in the COAT monitoring of the Arctic Tundra.

Using data provided by COAT, we will develop a prototype of virtual sensors. Users can interact with the virtual sensors through a user interface. The virtual sensors retrieves their data from the fused data from the data storage containing raw data and annotated data from the physical camera trap sensors.

Results show that ...

Contents

List of Figures

List of Tables

1 Introduction

The Arctic tundra in the far northern hemisphere is challenged by climate changes in the world today and is one of the ecosystems that are most affected by these changes [10]. The *Climate-ecological Observatory for Arctic Tundra - COAT* is a long-term research project developed by five Fram Center[1] institutions. Their goal is to create robust observation systems which enable documentation and understanding of climate change impacts on the Arctic tundra ecosystems. COAT was in autumn 2015 granted substantial funding to establish research infrastructure which allowed them to start up a research infrastructure during 2016-2020 [10].

Every year during the winter, COAT deploys several camera traps in eastern Finnmark, Norway for roughly one month. The main purpose of these cameras is to study predator populations, with especially a focus on the redlisted arctic fox which is the most endagered mammal species in Norway [9]. The camera traps are set up to take a time-lapsed photo every fifth minute during day and night which adds up to over 300 000 images per year [11]. Collecting this high volume of images gives Big Data challenges in the ecology field.

Managing the sensors and collecting data is currently a manual task that can introduce many errors and it is therefore a need for systems and abstractions which makes it simple to develop, maintain and use sensors and sensor data in the Arctic tundra.

1. http://www.framsenteret.no/english

1.1 Motivation

The motivation behind this project is that no single sensor may cover the sensing needs, and that sensing needs can change rapidly over time. Consequently, there is a need for sensor fusion, and allow for combining sensors at different computers.

This project will develop a prototype of virtual sensors. The purpose is to provide for a more powerful and flexible sensor in the COAT monitoring of the Arctic tundra.

Assume biologists want to search for image of a specific animal from different locations in the Arctic Tundra. A virtual sensor would locate raw data from the data storage and then return those raw data images that matches to the biologists request.

1.2 Contributions

The dissertation makes the following contributions:

- An introduction to virtual sensors.
- A description of data set preparation from the camera traps in the Arctic tundra.
- An implementation and description of a system that fuse raw data from different physical sensors into one virtual sensor.
- An evaluation of the system.

1.3 Limitations

We limit our approach to only use homogeneous data from physical sensors. This data is limited to be raw data images from camera trap sensors to simplify and limit the scope of the virtual sensor abstraction instead of supporting all possible data types. There is also a limitation to what a user can search for. We only implemented search for animals like red fox, raven and golden eagle or images containing no animals.

2

Related Work

Sensors are becoming omnipresent in everyday life and is generating data at an remarkable rate and scale [14]. Sensors and models play a vital role in harnessing "Big Data" to extract information. As more data become available from sensors, it will require rethinking of the scales, the processed models and creative thinking about fusing data.

The arrival of new instrumentation and sensors has led to an increasing amount of data which are becoming available for researchers and practitioners. However, accessing and integrating these data into usable environment for analysis and modeling can be highly time-consuming and challenging, particularly in real time [8]. As a solution to this challenge, Hill. et al. developed a prototype where users can exploit deployed virtual sensor types and interactively create and share new customized virtual sensors at different locations and suitable parameters to the researcher's desire. The system can provide point-averaged radar-rainfall products at either temporal resolution of the radar or as a temporal average of a fixed time period, which is illustrated in Figure 2.1 In contrast, our virtual sensors system is running on-demand, not in real-time. Our approach has neither the ability to provide users to create and share new customized virtual sensor.

A virtual sensor is a constructed sensor in constrast to a physical sensor [1]. They are used in place of the real sensors where they read real physical sensor data and calculate the outputs by using some processing models.

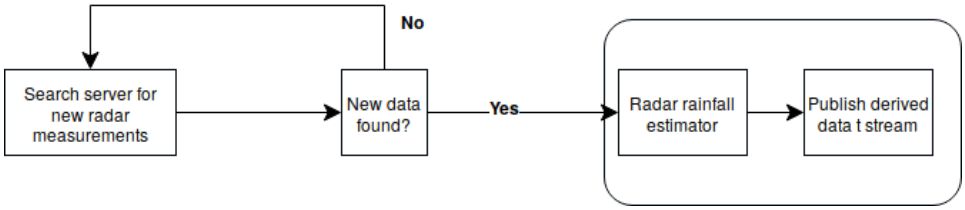


Figure 2.1: Figure illustrating the virtual rainfall sensor.

Previous research have focused on simple in-network data aggregation techniques and the sensor networks are often represented as a database. Two examples of such approaches are TinyDB [6] and Cougar [7], which enable applications to have a central point (base station) and create routing trees to funnel replies back to this root. The main focus on these approaches are operating intelligent in-network aggregation and routing to reduce the overall energy cost. In both examples, the data aggregation is specified using an SQL-like language. However, queries cannot be used to merge different data types, only homogeneous data aggregation is achievable. Their virtual sensors approach is offering a simple interface and heterogeneous in-network data aggregation. Our virtual sensors have no SQL-like language or a database to store the aggregated data. Our work also fuses physical sensor data which is already in a data storage, and not consuming directly from the physical sensors.

A virtual node[2] is a development of a set of physical sensors that a programmer can interact with as a single sensor. Their virtual nodes are implemented using TinyOS [3] and is specified using logical neighborhoods [4][5]. The nodes in the logical neighborhood are specified based on their characteristics by the programmer. Our approach of virtual sensors are not specified by the use of logical neighborhoods. There are no communication between the virtual nodes.

3

The Dataset

The dataset is provided by COAT and contains over 1.5 millions of pictures taken from 2011 to 2015 by their camera traps stationed in the Arctic Tundra in Finnmark, Norway. The wildlife camera traps from Reconyx are placed at six different areas in the Arctic Tundra: Nordkynn, Ifjord, Komag, Nyborg, Stjernevann and Gaissene. The dataset contains pictures during night- and daytime. The cameras have infrared flash so the cameras can take pictures at nighttime. These pictures are without color while the pictures taken at daytime are with color, as shown in Figure 3.1.

3.1 Dataset Directory Structure

The pictures in the dataset is structured in directories and folders. The dataset is first divided into different years going from 2011 to 2015. Each of these folders are then again divided into the areas of the 5 camera traps in the Arctic Tundra. Inside each of these folders specified by the area, they contain folders from each site inside the specific area. Figure 3.2 shows a cropped screenshot of the directories.

(a) Picture of ravens at daytime.

(b) Picture of an arctic fox at nighttime.

Figure 3.1: Images from the COAT dataset showing how a picture can contain colors or greyscale.

3.2 Image Annotations

COAT provided annotations stored in excel-files with information about all of the images in the dataset that biologist and ecologists working at the COAT project have detected. The biologist have used these annotations to see what animal was present at a specific time and to see statistics. The biologist have not made these annotations for the purpose to go back in time and look at the images, which is the essential part in our approach. The annotation contains information about a pictures location, site, date, time, animal classification, temperature in Fahrenheit, brightness, sharpness, saturation, sensitivity etc.

The excel-files are structured in their own folder and each file is named as a combination of the camera traps area, it's year and the site. An example is "*fotoboks2011_nordkynn_nordkynn.2011.xlsx*".

The dataset contains the images metadata and animal classification, but did not contain any filenames or a filepath. We then had to find a method to find out which images correspond to the metadata in the excel-files. Fortunately, each image had metadata stored in Exchangeable Image File Format (EXIF). We recursively traverse the directories where images are located and read date-time from the metadata and store it in a new CSV-file (Comma-separated Values). This was quite a time-consuming task because of the number of images to process.

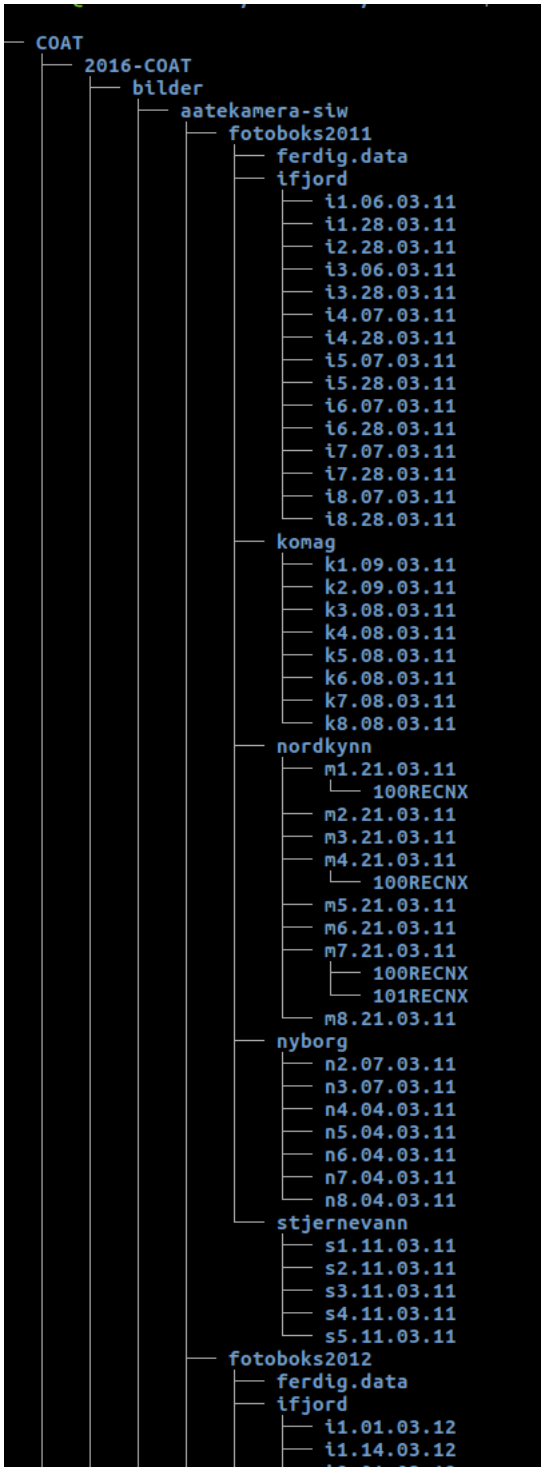


Figure 3.2: Figure showing a cropped screenshot of the structure of the directories.

4

Architecture

This chapter describes the architecture of the system. The components in the system are: physical sensors, raw data, analytics and classified data, fusion of data, fused data, virtual sensors and the user interface. The architecture of the system is presented in Figure 4.1. The arrows indicates the dataflow between each component in the system.

4.1 Physical Sensors

The physical sensors produce raw data. The raw data consists of images from different bait-camera sensors. The data also contains metadata about each image, as described in Chapter 3.

4.2 Raw Data, Analytics and Labeled Data

The raw data contains images from the physical sensors and annotations in excel-files containing metadata about each image alongside labeled data, as described in Chapter 3.

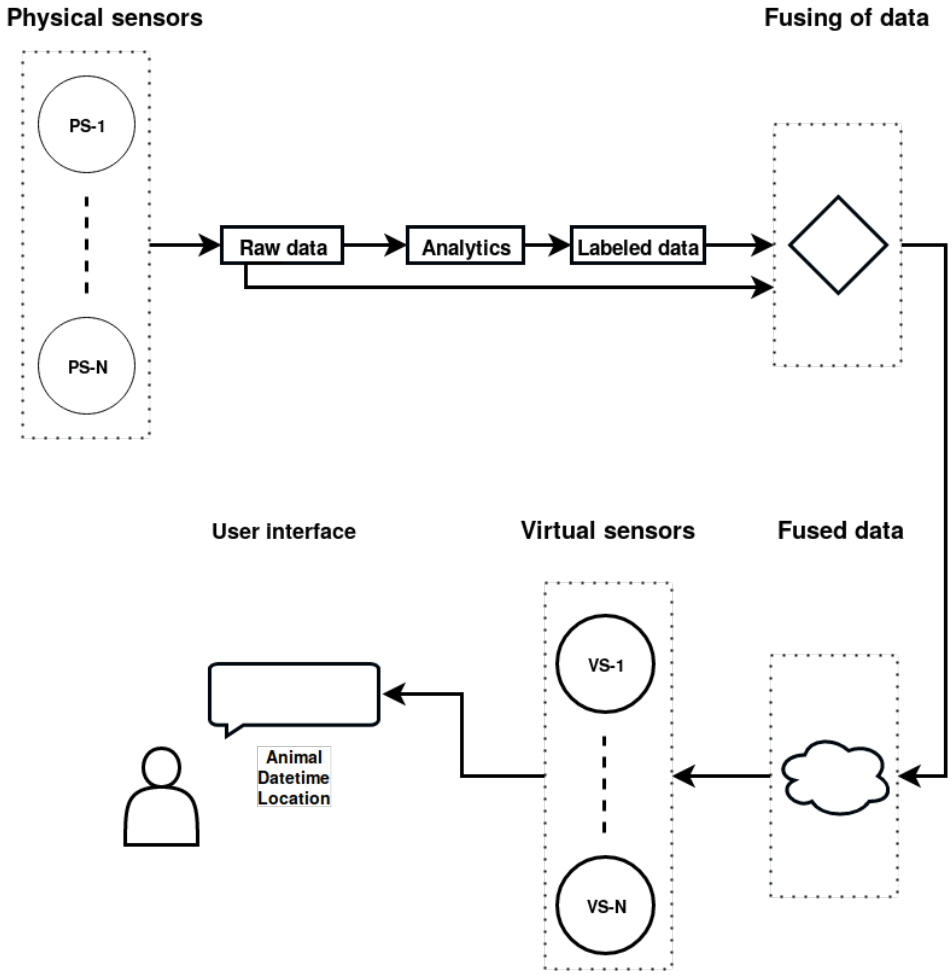


Figure 4.1: Figure showing the system architecture.

4.3 Fusing Of Data

Get analytics and classified data from the raw-data and annotated data. The raw data and the annotated data is compared and stored locally in the fused data.

4.4 Fused Data

The fused data retrieves it's data from the local storage. The fused data is the physical sensors location combined with necessary metadata such as date-time, location, site, year, animal classification and also how many animals there was in an image.

4.5 Virtual Sensors

The virtual sensors are divided into multiple sensors where each virtual sensor represents a different animal that scientists are interested in, like ravens, red foxes, golden eagles or polar foxes.

The virtual sensors are design identical and have the same functionalities by providing the ability to retrieve result from the fused data. The virtual sensors offers interaction between the fused data and the user interface. A user tells the user interface, described in Section 4.6, a task and forwards the task to the desired virtual sensor. The virtual sensor will then retrieve its results from the fused data.

4.6 User Interface

A user wants to retrieve fused data about animals. The user interacts with the virtual sensor user interface. This takes care of the interaction with the virtual sensor. When a user gives a command, the desired virtual sensor retrieve data from the fused data as described in Section 4.5, and presents the result back to the user.

5 Design

In this chapter we will look at the design of the system and present the design of each component of the architecture. Figure 5.1 shows the design of the system. The arrows indicates the communication lines and dataflow between each component in the system. The dotted arrows show how the data storage is structured with files and raw data.

5.1 Data Storage

The data storage contains raw data from the physical camera trap sensors and excel-files containing annotations for each raw data, as described in Chapter 3.

5.2 Sensor Fusion

As mentioned earlier in Chapter 3, the dataset did not contain any filename or a filepath. It only contained image metadata such as a pictures location, date-time, site, year and animal classification. We then had to find a method to find out which images correspond to which annotation in the excel-sheet.

We recursively traverse the directories where pictures are located and read

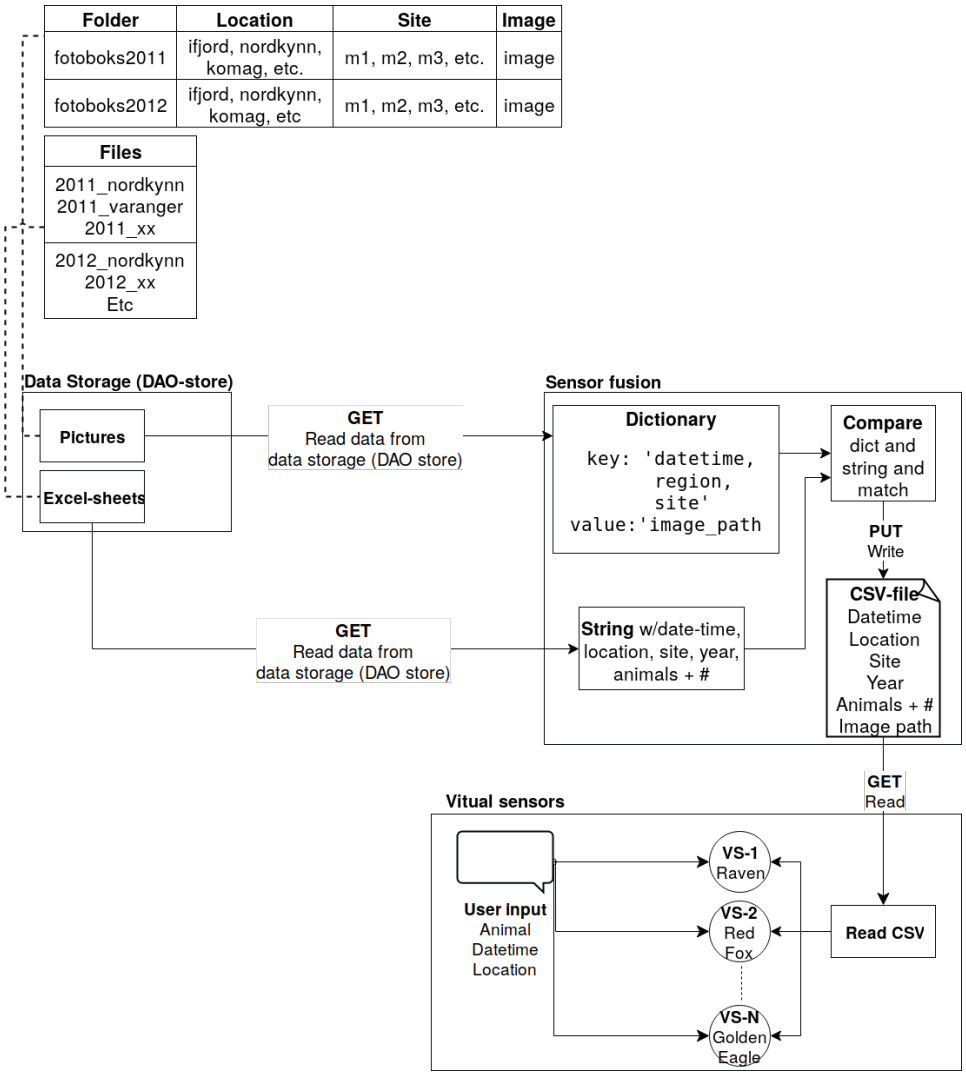


Figure 5.1: Figure showing the system design.

date and time from the metadata for each picture along with the annotation in the excel-sheet. The images and annotations that matched, is stored in a Python dictionary, like a key-value store, with date-time as key and image-path as value. Images and annotations that matches is stored in a new CSV-file. As mentioned earlier, was this quite a time-consuming task because of the numbers of images to process and compare.

5.3 Virtual Sensors

5.3.1 Virtual Sensors

The virtual sensors are divided into multiple sensors depending on animals. The respective virtual sensor must handle events from the user. Assume a user wants to find pictures of a red fox at a specific time. The user interface makes the respective virtual sensor aware of the task. The virtual sensor will then go to a list of all red foxes that are located from the fused data and search for the specific request. If the virtual sensor find pictures that are similar to the request from the user, one and one picture is visualized on the computer screen to the user.

5.3.2 User Application

The user-application is the connection between the user and the virtual sensors. It gets input from the user and sends a request to the virtual sensor based on its input. The user-applications goal is to make an interface that is easy to use and to show images that the user want to see. The user only need to specify the following 3 parameters for the virtual sensor:

- **Animal:** The user can choose to see 3 different animals; raven, red fox and the golden eagle. The user can also chose to see pictures with no animals in it. The reason that the user only can chose between these 4 categories is that these are the only virtual sensors that are implemented in this version of the system.

- **Date-time:** If the user wants a specific date-time the picture(s) was taken, he/she can specify the date and time in this section. If the user leave this field blank, all dates will included when the system search for pictures.

- **Location:** At last, the user can chose if he/she wants the picture to be located anywhere specific. The only valid arguments in this field is Nordkynn, Komag, Nyborg or Stjernevann since these are the only names in the CSV-file in the fused data.

6 Implementation

6.1 Overview

This chapter will elaborate on how we implemented the system, general implementation requirements, issues and choices. We will first look at some libraries used in this implementation, then we will take a look at the data storage in Section 6.2. At last, we will describe the virtual sensors and the user application in Section 6.3.

The system is implemented and written in the high-level programming language Python 2.7[1]. This was a practical choice because we were familiar with it from previous projects, it is an object oriented language and if offers multiple different libraries. The libraries used in this implementation are mentioned below.

Exchangeable Image File Format - EXIF

Exchangeable Image File Format (EXIF) is a standard that specifies the formats for images, sound, scanners and other systems recorded by digital cameras. This standard consists of image information of the EXIF image file or the EXIF audio file.

1. https://www.python.org/

Python have a module called EXIF 2.1.2[2], to extract the metadata(image information from the jpg-files.

Pandas Dataframe

Pandas[3] is an open source Python Data Analysis Library which provides high-performance and easy-to-use data structures. It is a Open Source sponsored project. A Pandas dataframe[4] is a two-dimensional, dynamic tabular data structure with labeled axes (rows and columns), just like an excel spreadsheet.

OpenCV

OpenCV V2.4.9.1[5] is an open-source library which provides the functionality for loading and preprocessing images.

6.2 Data Storage

The data storage contains raw data and preprocessed images that are structured and sorted by animal species. Unfortunately, these preprocessed images don't include image metadata so we don't have any information of when the image is taken. Therefore, the whole data storage is traversed to find the raw data with metadata and compare these images to the excel-files containing information like date, time, location and site.

6.2.1 Raw Data Information Extraction

The raw data in the data storage is traversed by walking the OS directories. If the traversed file ends with JPG, the image information in the EXIF is being extracted and the datetime value is read. This datetime along with the image's site, region and image path is stored in a Python dictionary, where the datetime, site and region is key and the image path is the value, as shown in Table 6.1. This was as mentioned earlier, a time-consuming task because of the 1.6 millions of pictures that was being processed.

2. `https://pypi.python.org/pypi/ExifRead`
3. `http://pandas.pydata.org/`
4. `https://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.html`
5. `https://opencv-python-tutroals.readthedocs.io/en/latest/`

Key		Value
Site	*Region*	*Image path*
m2	Nordkynn	/../2016-COAT/../nordkynn/m2.21.03.11/IMG_0172.JPG

Table 6.1: Show how an image's metadata is stored.

During implementation we came across a problem with the folders containing white-space and UTF-8 character. An example of a folder is *'/../Bilder til institutt for informatikk/åtekamera (Siw)'*.

This was an obstacle during implementation because it led to difficulties reading the folder-names when they contained characters the system wouldn't interpret. The solution was to change the folder-names manually and avoid the use of whitespace and UTF-8 characters. An example of a new folder is *'../bilder/aatekamera-siw'*. How this can be automated is described in Section 8.4.

6.2.2 Excel files Extraction

The excel files contains annotations for each image. To read excel files, we use Pandas Dataframe, described in Section 6.1. From the excel files, we extract information like date, time, animals, region, site and year. The necessary information about each animal is stored in a string.

During implementation we came across a problem with inconsistency between cells in the same row. The problem was that the datetime cells information was inconsistent with the information in cells day, month, year, hours and minutes, shown in Figure 6.1. Our solution was to alter and concatenate the correct values to make a new datetime.

Another problem was that the cells containing column-names was different. One example is that month was written "*moth*" in some excel files and "*month*" in others. Because of time constraints, we only used the excel files containing the word "moth" since this was the first excel-file to be extracted and our intent

23	3	2011	7	40	0	m7.11	T	3/22/2011
23	3	2011	7	50	0	m7.11	T	3/22/2011
23	3	2011	8	0	0	m7.11	T	3/22/2011
23	3	2011	8	10	0	m7.11	T	3/22/2011
23	3	2011	8	20	0	m7.11	T	3/22/2011
23	3	2011	8	30	0	m7.11	T	3/22/2011
23	3	2011	8	40	0	m7.11	T	3/22/2011
23	3	2011	8	50	0	m7.11	T	3/22/2011

Figure 6.1: Shows inconsistency between cells in rows in excel file.

was to get images to visualize.

A different problem was that some column-header names was written with a big letter like "*Time*" in some excel files and other contained "*time*". Our solution was to make every column header into lowercases using Pandas Dataframe function, *pandas.Series.str.lower()*.

JM: Her ville det nok vært en fordel med Python3!!!!!! One more problem arose when some of the excel-file contained table-headers with UTF-8 characters, as seen in Figure 6.2. These characters won't be interpreted by the system and the table-headers will not be readable.

6.2.3 Comparison of Data

The extracted metadata from the images and the annotations in the excel files that matches, is stored in a CSV-file (Comma-separated Values). We chose to store the data in a new CSV-file because of the large number of images to process. It will simplify search in the future since it will be faster to search through the CSV instead of searching through the raw data and compare them with the annotations. This is not the best choice, however it is a introduction to a solution.

6.3 Virtual Sensors

6.3.1 Virtual Sensors

The virtual sensors are implemented as functions, not threads or independent subprocesses, as mentioned in Section 5.3.1. The reason for this choice is because we needed somewhere to start and this was a simple approach. Lack of time prevented us to investigate other solutions with threads or subprocesses. Instead, the focus was to show images to a user and continue development as long as time didn't become an hindrance.

We also chose to focus on virtual sensors for three different animals and images without any animals in it. The three animals were the red fox, ravens and the

Figure 6.2: Shows table-headers with unsupported UTF-8.

golden eagle. We made this choice because these were the animals that were most present in the annotations in the excel files that were extracted.

The OpenCV library is used to visualize images to the user. The library read the images it is given and display these on the screen to the user. The reason for choosing this visualization library was because of experience with this in previous projects and also that our focus was not primarily on which library to use when visualizing images.

6.3.2 User Application

The user application is implemented as an interface that is visualized through the terminal window on the computer. The user interface is shown in Listing 6.1. The user application communicates with the virtual sensors by sending the users query to the corresponding virtual sensor. The corresponding virtual sensor is the sensor that contains the animal species the user specifies in the user query.

There is currently not possible for more than one user to be connected in the same application at a time. Because of lack of time, we have not implemented a solution for this. Our focus has been making a virtual sensor abstraction system and not the user interface. This user application is developed to be simple to use for the users and to show images on the users request.

Listing 6.1: Main menu

```
**** WELCOME! ****
******************

Write what you want to see
(Raven, RedFox, GoldenEagle):

Write when you want to see (blank is all dates):
E.g: 2011:03:24 or 2011:04:24 10:10:00

Where do you want the animal to be located
(nordkynn, varanger: komag, nyborg, stjernevann):
```

7 Evaluation

This chapter describes the experimental setup and metrics used to evaluate the implemented system.

7.1 Experimental Setup

All experiements were done on a Lenovo ThinkCenter with an Intel® Core™ i5-6400T CPU @ 2.20GHz × 4, Intel® HD Graphics 530 (Skylake GT2), 15,6 GiB memory and 503 GB disk. It ran on Ubuntu 17.04 64-bit with gcc V6.3.0 compiler and Python V2.7.13.

7.2 Experimental Design

We measured execution time to locate images in the datastorage and read the images metadata, read annotations from the excel-file and at last compare the images metadata with the annotations in the excel-file and write it to a new CSV-file through five experiments.

Three of these experiments where we compare image's metadata and annotations from the excel-file, were run concurrently with 4, 100 and 500 processes by Pythons multiprocessing library. The library allows fully leverage on multi-

ple processors on a given machine and supports spawning processes using an API similar to a threading module.

We also measured system utilization such as CPU, memory and disks by using Pythons psutil[1] library while running the system as described above. The functions used for measuring system utilization are listed below:

- **CPU - cpu_percent:** Return a float representing the current system-wide CPU utilization as a percentage.
- **MEMORY - virtual_memory:** Return statistics about system memory usage.
- **DISKS - disk_io_counters:** Return system-wide disk I/O statistics as a named tuple including the following fields:
  - **Read count:** number of reads.
  - **Write count:** number of writes.
  - **Read bytes:** number of bytes read.
  - **Write bytes:** number of bytes written

7.3 Results

In this section we will discuss the results of the testing described in the sections above.

7.3.1 Measure Execution Time

We measured time usage in minutes per task during system runtime to inspect which tasks that spends most time executing when running the system. We can see from Figure 7.1 that the compare-task takes the longest time followed by the part that finds folders and read metadata from the raw data. This is because of the of the high number of data to process. The reason that finding folder and metadata is relatively similar in each experiment is because it is the same number of images to traverse each time. This also applies to the execution time reading an excel-files.

1. https://psutil.readthedocs.io/en/latest/#

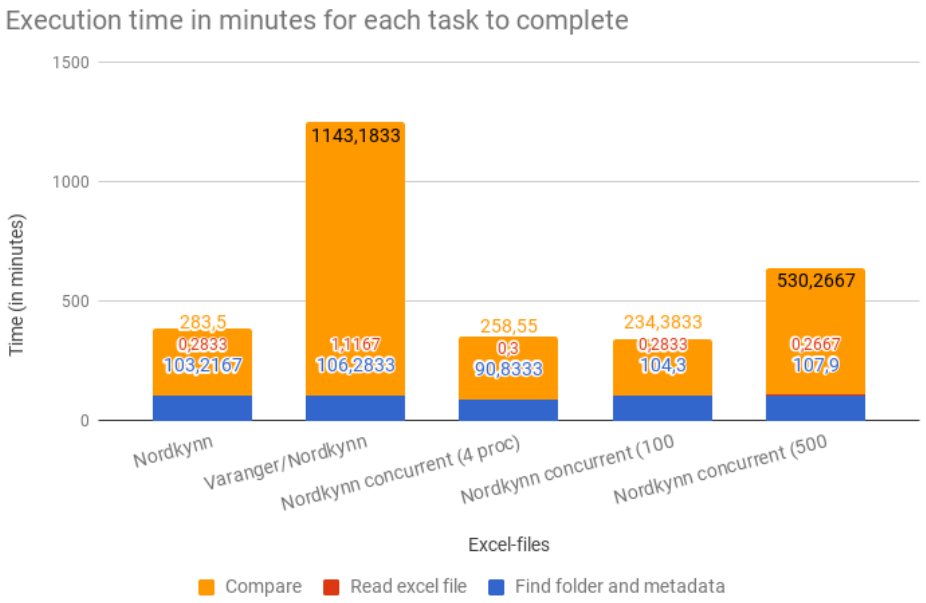


Figure 7.1: Figure showing time to execute different functions in the system (in minutes). These functions are the comparing between image's metadata and annotations in the excel-file(s), time to read the excel-file(s) and time used to find images and it's metadata.

The only bars in the figure that stands out from the other, is the comparison of data from both Varanger and Nordkynn. We can see that by increasing the data size by adding another excel-file, the time usage when comparing data escalate by a factor of 4. We can also see that the comparison with 500 processes stands out probably because over the overhead of running so many processes and I/O boundaries.

Figure 7.2 show the time usage for the whole system to run. As we can see, are the result similar to Figure 7.1. To read images and extract metadata, read one excel-file and compare the data, it takes approximately the same execution time with a sequential program and with 4 and 100 processes provided by the multiprocessing tool.

The system reads and writes to disk at an highly frequency and with the insight that hard drives in most cases are slow, we can draw the conclusion that the system is I/O bound. This can also explain why the multiprocess-implementations are relatively similar to the sequential implementation.

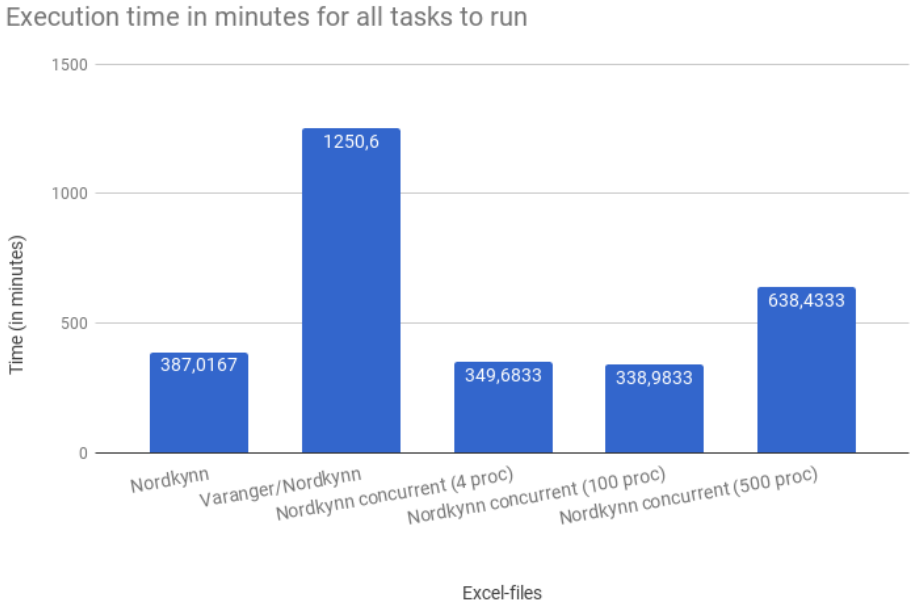


Figure 7.2: Figure showing time to execute the system (in minutes) from starting the system to the system is finished.

7.3.2 Measure CPU Utilities

We measured the CPU utilities to examine the CPU percentage of the system. As we can see in Figure 7.3, the CPU percentage peaks during operations. The system starts with traversing and read metadata from the raw data in the data storage which cause the peaks at the start of the graph. Comparing metadata and annotations in the excel-file is the last task. A which use low CPU percentage, as seen in the figure, and a reason for this may be that the CPU must wait because the comparing-task writes it's result to a file, as seen in Figure 7.6a, with an increasing number of writes during system runtime.

7.3.3 Measure Memory Utilities

We measured the memory utilities to examine the memory usage of the system. Figure 7.4 shows the result during system runtime. We can see that the system have between 90% and 96% memory usage. The reason for the high percent of memory usage around 94% is most likely caused by the comparing-task which have a increasing number of reads, shown in Figure 7.5a, and writes shown in Figure 7.6a.

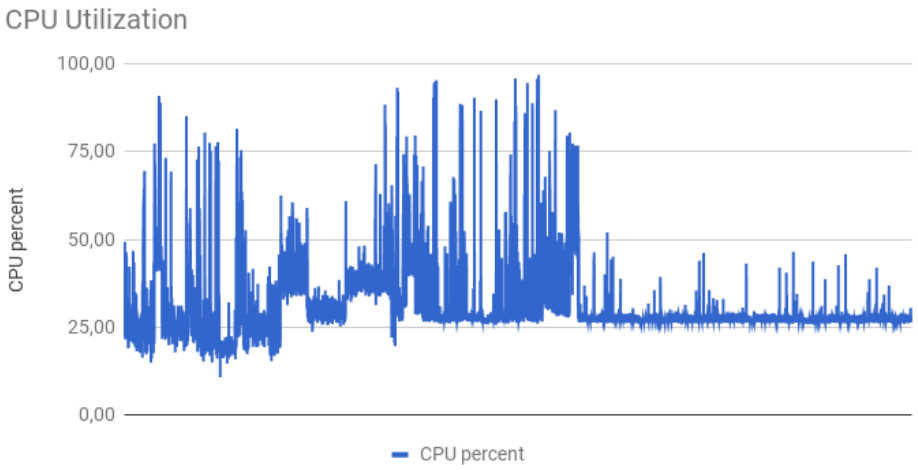


Figure 7.3: Figure showing CPU utilization if the system during runtime.

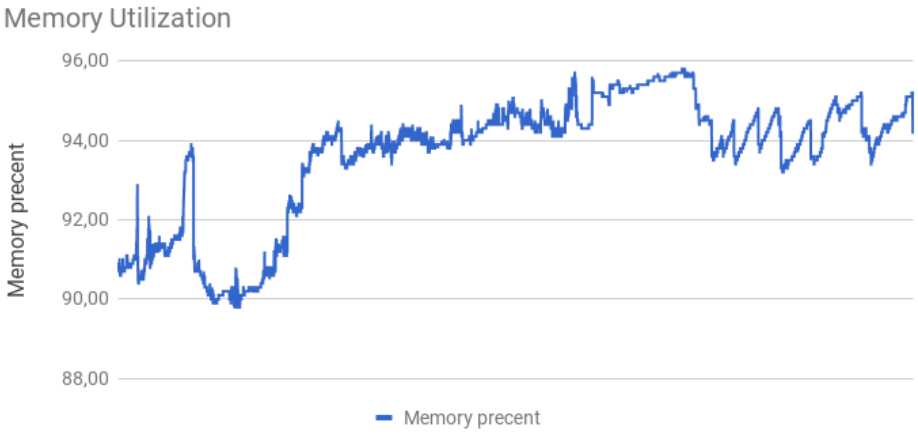


Figure 7.4: Figure showing memory utilization if the system during runtime.

7.3.4 Measure Disks Utilities

We measured the disk utilities to examine the disk I/O statistics of the system.

Read Utilization

As we can see in Figure 7.5a, increases the read count during system runtime. This is because of the comparing-task which is reading the values from the metadata and annotations.

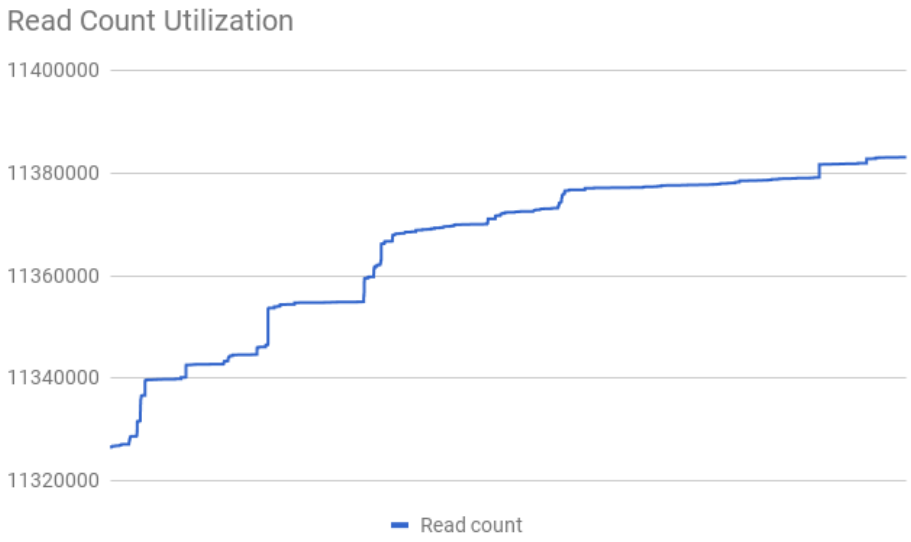
Figure 7.5b show number of reads compared to last read. As we can see, are there not many differences between a read and it's last read. There are some peaks during runtime, especially the peaks on a difference over 6000 reads. As we can see from Figure 7.5a, is there a peak vertically approximately at the same time. This is also a trend in the graphs in Figure 7.6.

kan se på antall counts at den går drastisk rett opp på samme tidspunkt.. noe som skjer der... Også peaks i read writes på samme tidspunkt..?

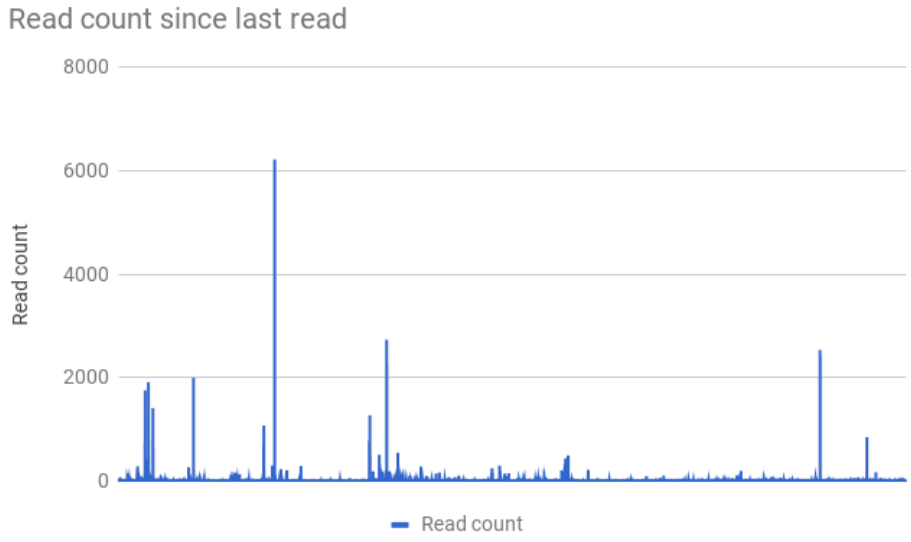
Write Utilization

As we can see in Figure 7.6a, there is an increasing number of writes during runtime. This is similar to the graphs showing read counts in Figure 7.5, since the coomparing-task is writing the result from the comparing in a new file.

Number of writes compared to last write is shown in Figure 7.6b. We can see that there are a few peaks during runtime

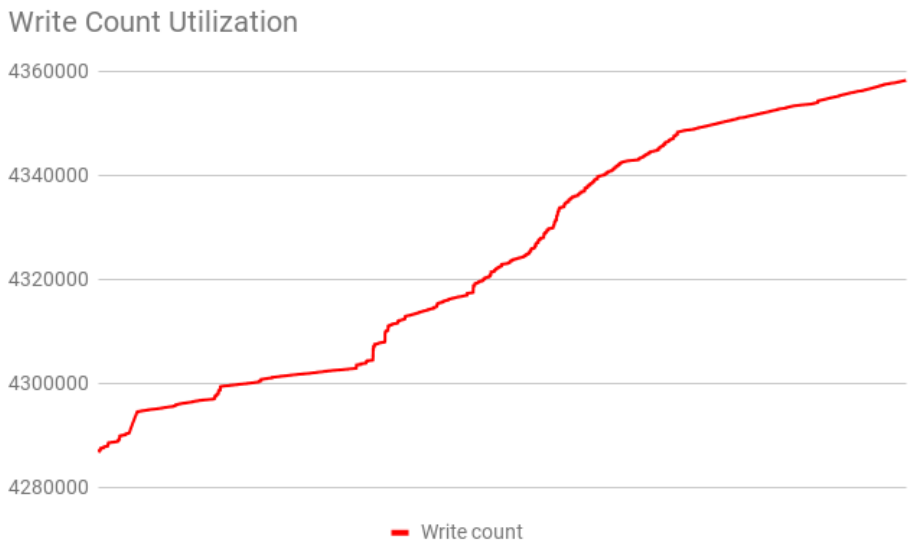


(a) Figure showing disk reads during system runtime.

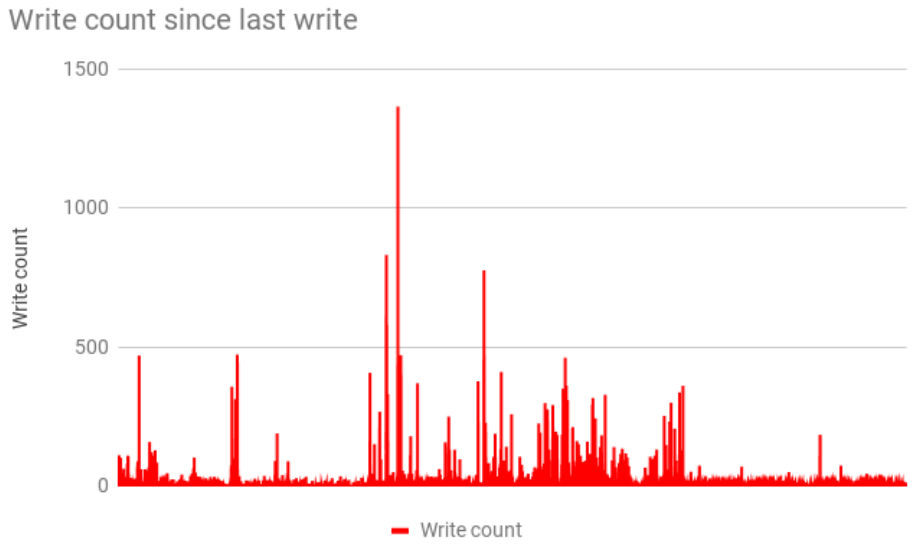


(b) Figure showing disk read compared to last disk read during system runtime.

Figure 7.5: Figure showing disk I/O reads during system runtime.



(a) Figure showing disk writes during system runtime.



(b) Figure showing disk writes compared to last disk read during system runtime.

Figure 7.6: Figure showing disk I/O writes during system runtime.

8 Discussion

This chapter discusses our approach, experience and why we chose the solution we ended up with.

8.1 Virtual Sensors

As mentioned in previous chapters, are the virtual sensors functions in our approach. We choose this approach because of we firstly wanted to find images from the fused data and show them to the users. Another reason is because of time limitations.

Another solution to this approach of virtual sensors is to implement them as threads. This would again not be the best solution if for example the system should be able to add more virtual sensors during run-time. To solve this problem, we could have implemented the virtual sensors as independent processes which would not be affected when another virtual sensors is added, removed or at worst crash/fail.

A virtual sensor could also be implemented as an abstraction or a general class with parameters such as *id, type, capabilities, history and queries or raw data*. This parameters will then say something about what kind of virtual sensor it is, what it can do (show images, find new data if any etc.) and support for queries from user.

If time wasn't a limitation, we would have tried to investigate the implementation of virtual sensors as an abstraction where a class with parameters specifies the different virtual sensors. These virtual sensors would also be subprocesses in the system so they don't affect each other.

8.2 Queries

Our approach support queries with only one option per animal, location and datetime, described in Section 5.3.2. The reason for this choice is because we started with a simple implementation with intent to expand at a later point. The first implementation only contained what kind of animal the user wanted to see. At later point, both location and datetime was added to narrow the search result. Due to time constraints, we didn't have time to implement more advanced search functionalities.

An improvement would then be to support operations like AND and OR for all three options in the user interface. Then the user could for example search for "*fox || raven*", "*fox && raven*", "*Nordkynn || Stjernevann*" or to choose an interval between two datetimes like "*26-04-2016 - 30-04-2016*". This would make image search more flexible and add more advanced search queries for the users.

8.3 Sequential vs. concurrent implementation

As described in Chapter 7, the execution time of system is not satisfying considering how many hours the system use to complete. Using Pythons multiprocessing library did not solve any problem with time spent on comparing images and annotations. The reason we tried Pythons multiprocessing library was to see if it made any performance-improvement on execution time compared to the sequential system. If it haven't been for the time constraints, we would try to implement another solution with threads.

Another solution is to change the language from Python to The Go Programming Language[1]. One of Go's strongest sides is a built-in concurrency based on CSP [13]. Go is built with concurrency in mind and has goroutines instead of threads. Another advantages of goroutines is that it comes with a built-in primitive to communicate between themselves by using something called channels. There is also no need for some kind of mutex locking when sharing data

1. `https://golang.org/`

structures compared to threads. However, there is no purpose of implement the system in another language if the system is I/O bound. To improve the system runtime, there is a need to implement a system with parallel tasks, like for example parallel comparing of image metadata and annotations.

8.4 Updates From Datastorage

Current solution does not support the ability to traverse the data storage with multiple excel-files. It only takes one and one excel-files that is predefined and extracts it's data.

A quick fix to this problem is to traverse the directory where the files are located and then check if the files ends with "*.xlsx*". These files will then be extracted and stored in for example a Pandas Dataframe like the current approach. The file-names could also be stored in the system to prevent to read the same files again if any new files is stored in the data storage.

We did not choose this approach because the comparing of images and annotations is a time-consuming task where one file took several hours and comparing with two excel-files took over 20 hours as described in Chapter 7.

Another challenge with this approach is that some data in the excel-files contains characters with UTF-8 encoding. The challenge is then to interpret the data, described in Section 6.2.2.

9

Conclusion

In this project, we have implemented a prototype of virtual sensors with the purpose to provide a more powerful and flexible sensor monitoring in the Arctic tundra. We give an introduction to virtual sensors and a description of the dataset from the camera traps in the Arctic Tundra. We describes a system that that fuse raw data from different physical sensors into one virtual sensor and provide an evaluation of the system. We've focused on showing results to the user and spend a lot of time with the dataset and modify the data so it could be extracted in the system.

Our experiments showed that the system have a long usage time during runtime and processing the high amount of data results in an I/O bound system. There is a need for some kind of parallel tasks or a concurrent system to improve performance on time usage.

10

Future Work

We will outline some of the areas that can be elaborated in future work. These include:

- **Implement virtual sensors as abstractions and subprocesses:** The implemented virtual sensors is our first prototype. As described in Section 8.1, would another approach be to implement the virtual sensors as abstractions and subprocesses. This would improve the virtual sensors by making it possible to create, change or remove sensors if needed. It would also give the user the ability to combine multiple virtual sensors to a new virtual sensor.

- **Query - combine data from multiple sensors:** This implementation of the system doesn't support multiple different search queries. In Section 8.2, we discuss how we could improve this by adding more advanced functionalitites to the user application. This would be an improvement because it would for example narrow a users search result when searching for "*red fox && raven*" or to retrieve images between to datetimes.

- **Automated data storage updates:** Adding an solution to an automated data storage update is discussed in Section 8.4. Current solution only supports on-demand updates and there is definitely a need for a script running in the background checking for updates in the data storage. This script should also be implemented to support a solution that reads all files containing annotation of data and store this in a list so we can check

if the file has been read or if it is a new one.

- **Improve execution time:** As discussed in Section 8.3, there is a need for improvement when comparing images metadata with annotations. A solution to improvement is to use concurrent computing and investigate this solution using subprocesses, threads, another concurrency control library or a parallel implementation.

11

Appendix?

readme, source code, dataset measurement RAW

Bibliography

[1] S. Kabadayi and A. Pridgen and C. Julien, *Virtual Sensors: Abstracting Data from Physical Sensors*, 2006, in *2006 International Symposium on a World of Wireless, Mobile and Multimedia Networks(WoWMoM'06), 6 pp.-592*.

[2] Ciciriello, Pietro and Mottola, Luca and Picco, Gian Pietro, *Building Virtual Sensors and Actuators over Logical Neighborhoods*, 2006, in ACM, 19–24. *http://doi.acm.org/10.1145/1176866.1176870*.

[3] Hill, Jason and Szewczyk, Robert and Woo, Alec and Hollar, Seth and Culler, David and Pister, Kristofer, *System Architecture Directions for Networked Sensors*, 2000, in *ASPLOS-IX: Proc. of the 9 nt Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 93–104.

[4] Mottola, Luca and Picco, Gian Pietro, *Logical Neighborhoods: A Programming Abstraction for Wireless Sensor Networks*, 2006, in *Distributed Computing in Sensor Systems: Second IEEE International Conference, DCOSS 2006, San Francisco, CA, USA, June 18-20, 2006. Proceedings*, pages 150–168. *https://doi.org/10.1007/11776178_10*.

[5] Mottola, Luca and Picco, Gian Pietro, *Programming Wireless Sensor Networks with Logical Neighborhoods*, 2006, in *Proceedings of the First International Conference on Integrated Internet Ad Hoc and Sensor Networks*, series = InterSense '06, pages 150–168. *http://doi.acm.org/10.1145/1142680.1142691*.

[6] Madden, Samuel R. and Franklin, Michael J. and Hellerstein, Joseph M. and Hong, Wei, *TinyDB: An Acquisitional Query Processing System for Sensor Networks*, March 2005, in *ACM Trans. Database Syst. Vol. 30, Nr. 1*, pages 122–173. *http://doi.acm.org/10.1145/1061318.1061322*.

[7] Yao, Yong and Gehrke, Johannes, *The Cougar Approach to In-network Query Processing in Sensor Networks*, September 2002, in *SIGMOD Rec., Vol. 31, Nr. 3*, pages 9–18. *http://doi.acm.org/10.1145/601858.601861*.

[8] David J. Hill and Yong Liu and Luigi Marini and Rob Kooper and Alejandro Rodriguez and Joe Futrelle and Barbara S. Minsker and James Myers and Terry McLaren, *A virtual sensor system for user-generated, real-time environmental data products*, 2011, in *Environmental Modelling & Software, Vol. 26, Nr. 12*, pages 1710 - 1724. *http://www.sciencedirect.com/science/article/pii/S1364815211001988*.

[9] Ims, R.A., Jepsen, J.U., Stien, A. & Yoccoz, N.G. 2013, *Science plan for COAT: Climate-ecological Observatory for Arctic Tundra. Fram Centre Report Series 1*, 2013, in *Fram Centre, Norway, 177 pages. http://www.framsenteret.no/getfile.php/2435814.1574.xyxruwywpp/FinalPDF_COAT.pdf*.

[10] Åshild Ø. Pedersen, A. Stien, E. Soininen, and R. A. Ims, *Climate-ecological observatory for arctic tundra-status 2016*, Mars 2016, in *Fram Forum 2016, pages 36-43.*

[11] S. Hamel, S. T. Killengreen, J.-A. Henden, N. E. Eide, L. Roed-Eriksen, R. A. Ims, and N. G. Yoccoz, *"Towards good practice guidance in using camera-traps in ecology: influence of sampling design on validity of ecological inferences*, 2013, in *Methods in Ecology and Evolution, vol. 4, no. 2. pp. 105-113*

[12] Gună, Ştefan and Mottola, Luca and Picco, Gian Pietro, *DICE: Monitoring Global Invariants with Wireless Sensor Networks*, June 2014, in *ACM, vol. 10, no. 4. pp. 54:1–54:34 http://doi.acm.org/10.1145/2509434*.

[13] Hoare, C. A. R., *Communicating Sequential Processes*, Aug. 1978, in *ACM, vol. 21, no. 8. pp. 666–677 http://doi.acm.org/10.1145/359576.359585*.

[14] Stefan Reis and Edmund Seto and Amanda Northcross and Nigel W.T. Quinn and Matteo Convertino and Rod L. Jones and Holger R. Maier and Uwe Schlink and Susanne Steinle and Massimo Vieno and Michael C. Wimberly, *Integrating modelling and smart sensors for environmental and human health*, 2015, in *Environmental Modelling & Software, vol. 74, no. Supplement C. pp. 238 - 246 https://doi.org/10.1016/j.envsoft.2015.06.003*.