University of Twente

Design Report

Faculty of Electrical Engineering, Mathematics & Computer Science

OpenINTEL Dashboard

Authors:
Chris Admiraal
Bart Meyers
Ruben Horck

Client / Supervisor:
Raffaele Sommese

November 12, 2021

Contents

1 Introduction 3
1.1 Motivation 4
2 Requirements 5
2.1 User Requirements 5
2.2 System Requirements 6
2.2.1 Must 6
2.2.2 Should 6
2.2.3 Could 7
3 Research 8
3.1 Back-End Research 8
3.1.1 General Idea 8
3.1.2 Back-End Implementation Options 8
3.1.3 Back-End Implementation Considerations 10
3.2 Front-End Research 11
3.2.1 General Idea 11
3.2.2 Front-End Implementation Options 11
3.2.3 Front-End Implementation Considerations 12
3.3 Conclusion 12
4 Design 13
4.1 Design Description 13
4.2 Design Choices 14
4.2.1 Adding data and graphs 14
4.2.2 Configuration 15
4.2.3 Data collection 15
4.2.4 Historical and live data collection 17
4.2.5 Public access 18
4.2.6 Security 18
5 Implementation 20
5.1 General system 20
5.2 Data retrieval 20
5.3 Data processing 20
5.4 Writing data to InfluxDB 21
5.5 Docker 21
5.6 Logging 22
5.7 InfluxDB 22
5.8 Grafana 22
6 Testing 24
6.1 Unit Tests 24

6.2 System Tests 24
7 Performance 25
7.1 Querying, transforming, and writing data 25
7.2 Dashboard performance 26
7.3 Conclusion 27
8 System Usage 29
8.1 Config files 29
8.1.1 Main Config 29
8.1.2 Table Config 29
8.1.3 Data Config 29
8.2 Manual 30
8.3 Provided Jupyter Notebook Python code 30
9 Discussion & Conclusion 31
9.1 Conclusion 31
9.1.1 Must 31
9.1.2 Should 31
9.1.3 Could 32
9.2 Discussion 32
9.3 Individual Contributions 33
9.4 Future Implementation 34
A Definition 35
A.1 Acronyms 35
A.2 Glossary 36
B Bibliography 37
C Appendix 38
C.1 Data config 38
C.2 Class diagram 41
C.3 Meetings 42

1. INTRODUCTION

OpenINTEL is a project that actively collects Domain Name System (DNS) data of many Top-Level Domain (TLD) and smaller domains, such as Country Code Top-Level Domain (ccTLD), e.g. *.nl* - for the Netherlands.
The DNS is a system the internet utilizes to assign names to its connected participants, namely computers or other resources. OpenINTEL collects time based data on statistics related to DNS queries daily, such as what type of DNS record is queried.
The reason for the collection of this data is that the DNS is a crucial part of nearly all internet services, by storing and reviewing these data-points one is able to track the global progression of internet services and state of the overall internet over time.
Because of the size of the internet and its connected participants, although not all need an assigned name, the result of this data collection is a massive amount of data. The research group overseeing OpenINTEL, named Design and Analysis of Communication Systems (DACS), has tasked us with creating a publicly accessible dashboard to showcase graphs on the statistics of the data that has been collected, as currently there exists no visualisation for said data. This dashboard should contain graphs and other visualisations to give a better perspective on the collected data, rather than having to interpret the data row by row as it is stored in the current storage. Using these visualisations, one should get a much faster insight into how certain DNS elements evolve over time.

1.1. Motivation

The reason for the collection of this data is that the DNS is a crucial part of nearly all internet services, by storing and reviewing these data-points one is able to track the global progression of internet services and state of the overall internet over time.
The motivation for the implementation and usage of a dashboard is based on the need for a tool to provide quick insights into the state the DNS and to use for assessing research. By using the possible graphs and other visualisations in the dashboard both of these requests can be fulfilled.
Furthermore the publication of the dashboard would remove some of the worries that; although currently OpenINTEL queries data from the global DNS network, it does not have a large impact on the DNS system. However, a lot of people or groups could want to have insight into similar data, since this data is so valuable in the global perspective. If sufficient people - who would need access to similar data - would query the same data, this might affect the system in a notable fashion. Therefore, creating a way to share collected data with people who could be interested, such as other researchers or journalists, is warranted. This place would be the dashboard. OpenINTEL is currently open to requests for data access, as per their website. This dashboard could also be used by other parties to gauge whether the OpenINTEL data would be suitable for their needs.

2. REQUIREMENTS

We have formulated the following the user- and system-requirements in communication with the client.

2.1. User Requirements

1. **The dashboard needs to show statistics about OpenINTEL data.**
   The data available in the dashboard should (indirectly) be coming from the existing OpenINTEL database.
2. **The dashboard needs to show graphs which are updated daily.**
   Once an admin adds a new graph, this graph should be updated daily to obtain the newest data.
3. **The dashboard needs to be public.**
   Anyone on the internet should be able to view the dashboard. But should of course not be able to edit this dashboard or the data.
4. **As an admin I want to be able to add different graphs/statistics.**
   The dashboard should be extendable with new graphs at any point in time. Only admins are allowed to do this. There should be a wide variety of visualizations that could be chosen.
5. **As an admin I want to be able to add historical data to the system.**
   When adding a new graph to the dashboard, the admin should be able to specify which historic data should be included.
6. **As a user I should be able to select a time period for the graphs.**
   When viewing the dashboard, the user should be able to apply a global time filter which filters all graphs to this selected time range.
7. **As a user I want to be able to filter on a TLD for the graphs**
   When viewing the dashboard, the user should be able to apply a global TLD (drop-down) filter which filters all graphs to the selected TLD(s).
8. **As an admin I want to be able to have insights in the running program.**
   The admin wants to see a log file and additionally be notified when something went wrong in the program.
9. **As a user I want to be able to travel back to the OpenINTEL website using a hyperlink.**
   The dashboard should include a link back to the OpenINTEL website so that the end user can surf to it.

2.2. System Requirements

For listing the requirements we will make use of the MoSCoW-methodology. In this methodology the requirements are categorized by the following keywords: "Must, Should, Could, Won't" (MSCW). Defining a requirement as one of these simply means the 'The design ... contain the following'. The requirements will be created using the SMART principle. This principle specifies that requirements need to be specific, measurable, acceptable, realistic, and time bound.

2.2.1. Must

1. **The system back-end must require authentication**
   The data stored on the back-end could include data that the public is not allowed to see. Furthermore the back-end could have a connection to other servers that should not be available to the public.
2. **The system must process newly added data after 3 days**
   The OpenINTEL DNS data is collected daily. However, this data is only fully available after about 3 days, after which the system should process it and make the results available for the dashboard. This period should be configurable.
3. **The system must collect the data from the HDFS**
   Hadoop Distributed File System (HDFS) is a technology used by OpenINTEL for storing their collected data. The system should be using this data.
4. **The system must be structured so that dashboard can be expanded by a few simple interactions**
   Adding new graphs is an important feature and should not be difficult.
5. **The system must be able to store the time series data acquired from the OpenINTEL cluster**
   There needs to be some kind of time series database, into which the results from the daily queries are stored in order to show them in the dashboard.

2.2.2. Should

1. **The dashboard should have quick loading times**
   The dashboard needs to be responsive, so that within 10 seconds one year of data for can be shown to one user in case only that user is requesting.
2. **The system should be able to handle future data**
   In the future more data might be collected, which might have a slightly different structure (e.g. more columns). The system should be designed such that adding this new data does not require further development.

3. **The system should be documented, so that an outsider could expand it**
   The documentation should be clear on how to expand the system with e.g. new graphs with daily queries etc.
4. **The dashboard should handle different kind of data**
   Different graphs might use different datasets which are differently structured. This also includes that the data should be handled case-insensitively.
5. **The dashboard should be able to show different kind of visualizations**
   Visualizations like line, bar, pie charts or statistics should be able to be chosen.
6. **The system should be able to process historic data**
   When data is added there should be the option to backfill the database with historic data.
7. **The database and front-end should run in a Docker container**
   Deploying the database and front-end should be done via Docker.
8. **The system should log its actions**
   The program should log the important operations to a single log file. Also, errors should be included.
9. **The dashboard should allow filtering**
   The graphs on the dashboard need to be able to be filtered on time and TLD.For the metrics issue a reverse proxy could be used to prevent users access. This reverse proxy could then also be used for the Hypertext Transfer Protocol Secure (HTTPS) connection.
10. **The system should handle SQL queries with admin defined table names**
    The admin defined table names are mapped to one or more parquet-paths in a configuration file. Any Structured Query Language (SQL) query should then be able to use these table names as described.

2.2.3. Could

1. **The rest of the system could run in a Docker container**
   Other parts of the system could also be included to run in Docker.
2. **The system could send a notification when an error occurs**
   Any error in any part of the program should trigger a notification which will send a mail to the admin.
3. **The dashboard could be styled like the OpenINTEL website**
   The dashboard needs to look like it belongs on the OpenINTEL website. This would allow the dashboard to be integrated into the website in the future.

3. RESEARCH

Before we started implementing, some research into our options was necessary. In this section we first explain our idea, after this we describe the research that was done to research aspects of this idea.

3.1. Back-End Research

3.1.1. General Idea

The general idea for the data visualisation was that data would be retrieved from the Hadoop Distributed File System (HDFS) using Spark and then to then save this in a Time Series Database (TSDB). A TSDB is used, as this type of database is specifically designed to store times and values. After this is achieved the data could then be queried from the database inside a front-end graph visualisation solution, so that unique graphs could be created inside that solution. These graphs could then later be exported as a whole or on a website.

Data Retrieval

The data retrieval would be done using Python. The decision to use Python was based mostly on our experience and the ease of use. Additionally, the system does not require a language that allows for optimizations in speed and memory use. The data itself would be retrieved from the HDFS using the PySpark [1] library, since the HDFS is the most convenient point of access - as recommended by the client. PySpark is capable of retrieving data from the HDFS using both Structured Query Language (SQL) queries as well as its own method-based query system.

This data would then be transformed into a PySpark DataFrame, allowing it to be written to the database. A Jupyter Notebook was provided by the client in week 2 of the project. As this notebook was coupled to a training data set, it could be used to get familiar to both PySpark and the data sets.

3.1.2. Back-End Implementation Options

After some research into Time Series Databases four possible options were exposed: InfluxDB, Elasticsearch, TimescaleDB and OpenTSDB. These had the ability to retrieve and store data in a time-based format, that would then enable us - later on in the process - to query data from specific time-spans.

To make a decision; research was conducted into all four database solutions. Next to the research, three of the four options were also tested in demo set-ups.

This all was more or less done in the second week of the project, since the plan was to get started on the actual implementation in the third week. The options were therefore limited to these four - either because they were recommended or they seemed like suitable alternatives after having done some research - so that we were able to make a decision sooner rather than later.

Back-End Option: InfluxDB

Research into InfluxDB was started as early as the second week of the project, as the client had described this solution in his introductory e-mail. It was found that InfluxDB's Python documentation was quite lacking. However, their documentation did give quite good examples on how to write large portions of data effectively. InfluxDB also uses another query language than SQL, called flux. This query language was unique and unfamiliar to us. Setting up InfluxDB seemed relatively simple and there is also a Docker image available. This assumption in setup simplicity would be put to the test in the two demos that were done for InfluxDB.

In these InfluxDB demos it seemed that the query language was not as big a problem as was expected, InfluxDB provides a web interface which can be used to both inspect your 'buckets' (a collection of datasets) as well as the individual measurement sets contained in these. Using the interface one could also let a small query be built. The query editor also provided drag-and-drop functionality, to add more statements into the generated query. These statements would however still need to be filled with proper parameters.
Furthermore, InfluxDB offers built-in control through which users (using authentication tokens) could read and/or write, this would be beneficial to portion off reading and writing for security reasons.

Back-End Option: Elasticsearch

Elasticsearch is part of the Elastic Stack, which meant that only along with the use of Kibana as the front-end solution - which is also part of the Elastic Stack - it could be used to its full potential. For Elasticsearch (and Kibana) two demos were created to get a grasp of how things would function using a more hands-on approach than simply researching their dedicated and other alternative websites.
During these demos we found that Elasticsearch was relatively simple to set-up and had a Docker image readily available. After the set-up it was recognized that Elasticsearch had a slow start-up and used quite a lot of system resources. Even idle Elasticsearch would often almost have the Random-Access Memory (RAM) capped out, this seemed mainly because of the overload of functionality that is provided out of the box. Strangely though, there seemed to be no authentication enabled by default. The data in Elasticsearch could be queried without any actual query writing using their web User Interface (UI). Elasticsearch, same as Grafana, had

multiple client frameworks available, including Python, although the documentation seemed quite outdated.

Back-End Option: TimescaleDB

During research into TimescaleDB it was reckoned to be a good contender for InfluxDB. InfluxDB had created flux as they felt support for time scale databases was limited in traditional SQL. TimescaleDB however still used a type of SQL (PostgreSQL), which would have been more convenient to use - as we project members were unfamiliar with flux. We assumed using a unique query language might also not be completely in line with the client's requirement to make things more expandable, although when we asked about this next meeting it turned out that the client was okay with a unique query language such as flux.

Whilst trying to create a demo it swiftly became quite apparent that TimescaleDB was heavily cloud and subscription focused (their web UI was - for example - behind such a subscription). The standalone version they provide was appeared to be a fairly simple extension to a normal PostgreSQL installation and was incredibly tedious to set-up. The process of querying data was how one would expect from any SQL based solutions, although it seemed like more extensive queries were required for querying the same data as the flux examples did in the InfluxDB demos.

Back-End Option: OpenTSDB

OpenTSDB is an option that uses Hadoop and HBase; and should basically contain one or more Time Series Daemons that all write to Hadoop independently (without master). It seemed to have very low latency and was extendable to write at 20 or 30 times the speed of InfluxDB, OpenTSDB could also use Hadoop files as direct input.

3.1.3. Back-End Implementation Considerations

During one of the project group's morning meetings the findings were discussed, how they would fit the requirements and personal preferences.

OpenTSDB was disregarded due to it appearing to be more or less a cluster solution, rather than something that could be used in once instance, because of this no attempts for a demo were made either.

For the back-end the main concerns were the speed - mainly readability - and the solution's support for Python; as these - was assumed - would be the bottlenecks of the back-end side of the proposed data pipeline. Python needed to function properly to write data at an effective rate, at least to be done with a day's data within a day of processing time. Faster would be better, as past data would also have to be written. Readability needed to function properly, because of the requirement to serve the end user with (adjusted) graphs in a responsive fashion.

3.2. Front-End Research

3.2.1. General Idea

The general idea pertaining the front-end is that graphing software would be used, in combination with the back-end solution, from which the front-end solution would receive the data. This graphing software should be capable of retrieving and visualising data from Time Series Databases such as Elasticsearch, InfluxDB, OpenTSDB and TimescaleDB.
A website or the graphing software's dashboard would then be used to showcase the data, this website should be styled similar to the OpenINTEL website. A user should not be able to alter back-end data, should be able to select a time range and preferably have some more options related to graph visualization, such as filters.

3.2.2. Front-End Implementation Options

For front-end graphing software our options seemed to be split between Grafana and Kibana. Grafana is more compatible with different back-end solutions than Kibana. Kibana is part of the Elastic Stack and therefore specifically meant to work with Elasticsearch. Below we go into more detail on these options.

Front-End Option: Grafana
One of our possible front-end graph visualisation options was Grafana, this solution was also recommended to us by the client. Nevertheless, research was still conducted into the option along with incorporation into the demos, so that we could come to a well-founded decision.
In the research we found that Grafana was overall more compatible with different back-end options such as InfluxDB, TimescaleDB and OpenTSDB. It would offer version control, enough customizability concerning graph creation and offer a way to export single graphs or the entire dashboard in an elementary fashion.
During our demos it was deduced that Grafana had a straightforward set-up and when coupling it to a database one would only need to fill-in the required credentials, after which a connection would be readily established. After establishment of the connection, one was able to query the data using a query editor, all inside a web-browser accessible Graphical User Interface (GUI). Grafana also offers support for custom time ranges and global dashboard variables that could be implemented throughout the solution.

Front-End Option: Kibana
Research into this option made quickly apparent that Kibana could only be considered if we chose for Elasticsearch as our back-end option, as they together were part of the Elastic Stack, hence together they were meant to work in perfect harmony.

During the demos it was found that the set-up was practically as smooth as the one for Grafana described above, it could all be done through a Graphical User Interface and Kibana has an available Docker image as well.
Kibana offers a very large range of visualisation options, along with a time-range and global filter. The whole dashboard could be shared to an external site using a snapshot.

3.2.3. Front-End Implementation Considerations

Since Kibana and Elasticsearch were in point of fact only properly compatible with one another and not so much with our other options, we figured that we should determine the best fit for our front-end/visualization tool first. As, if this happened to be Kibana our choice would have been limited to Elasticsearch for our back-end. The considerations with respect to the front-end were mainly focused on the ease of use, visualisation options, how well data in the front-end application would be disconnected from the back-end, the complexity of the set-up and integration with the database.

3.3. Conclusion

It was decided that TimescaleDB would be too complex in terms of usage, as - at most - it offered command line functionality. Their web UI features, which we would have preferred, were locked behind a subscription pay wall. Next to this the lack of functionality - it seemed more traditional, as an extension to a relational database. Therefore, it was thought to be not suitable as the other options.
The choice between Elasticsearch & Kibana versus InfluxDB & Grafana was made based on our preference for the front end. This preference was Grafana as it had more flexibility in the means of how the data could be exported - which was not completely thought out (designed) yet. Grafana seemed to have higher speed and stock-enabled - seemingly more secure - authentication. Kibana would have been simpler to use, but it was concluded that many of the offered features would not be required. These redundant features came at the cost of a much higher resource usage when using both Elasticsearch and Kibana. Hence, we decided to choose InfluxDB for data storage and Grafana for the data visualisation.

4. Design

4.1. Design Description

In figure 1 an overview of the proposed system can be seen. It shows the data flow from the OpenINTEL data cluster to the dashboard that is publicly accessible. The data is retrieved from the OpenINTEL cluster, through a combination of Python and SQL; using PySpark and a SparkContext to access and read from the HDFS using Apache Spark.
This data is retrieved once per day, per SQL query. Transformation of the data mostly happens during the data retrieval itself. After the data has been queried, a single query's resulting PySpark dataframe is used to fill a measurement in InfluxDB, the Time Series Database. This measurement can then be used by the queries of one or more Grafana graphs to visualize this data.

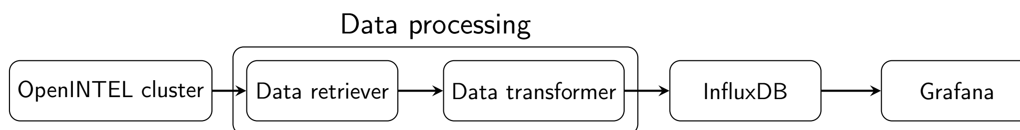

Figure 1: *High level system overview*

In figure 2, a more detailed overview of the system can be seen. This design was chosen based on the choices described in the next section.

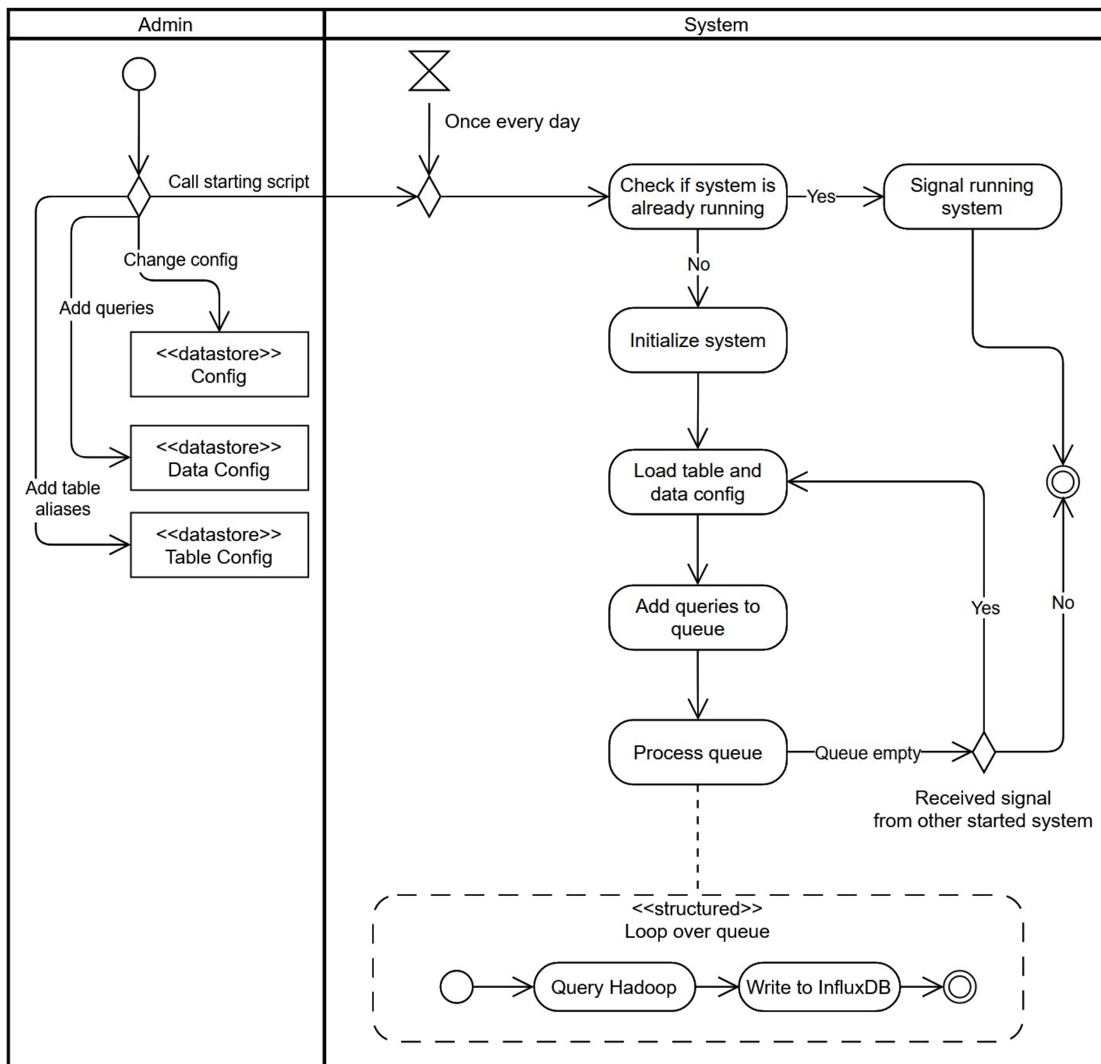

Figure 2: *Activity Diagram*

4.2. Design Choices

For the design several decisions needed to be made. In this section the options and decisions are discussed.

4.2.1. Adding data and graphs

One set of design choices was related to the requirement that graphs should be easy to add. Since Grafana was already chosen as the front-end and Grafana has a GUI to create and change dashboards the decision was made to use this. The reasoning was that creating a custom front-end would not improve functionality and take away time from other functionality.

Another part of making graphs easy to add is the method by which data can be added to InfluxDB. For this it was decided to use a configuration file. This would allow the client to add a query to this file, which would then be collected by the system. For the format of the file two options were considered. The JavaScript Object Notation (JSON) and Initialization File (INI) file formats. For both there is a Python library to read from these file formats. The JSON format has more flexibility than the INI format. For example, JSON supports lists while INI does not. INI files however are easier to read. Since the flexibility of JSON is not needed and ease of use is important the INI file format was chosen.

OpenINTEL has multiple tables in their Hadoop cluster. To make the system more flexible a method was needed to name these tables and reference them in the queries. Since INI files were already used for the other configuration files it was decided that the table configuration would also use this format.

4.2.2. Configuration

The system requires configuration to work. Settings such as how and where to log, where InfluxDB can be reached and the token to be able to write to InfluxDB. To make sure the client does not need to read and change the code, when for example InfluxDB is moved, it was decided to put these configuration settings in a configuration file. For the same reasons as in section 4.2.1 the INI file format is chosen here.

To ensure the system runs the intended way and not stop after a long period due to missing configuration, a configuration file checker was implemented. This way if required configuration is missing, the system will not start. For this, two options were considered. The first option was to hard code the setting that need to be configured. The second option was to compare the configuration file with a default configuration file. Assuming the default configuration has all required settings, this file can be used for verification. The second option is more flexible when adding more keys and was therefore chosen for the system.

4.2.3. Data collection

For the collection of data there were multiple decisions to be made. The first was on how to retrieve the data from the OpenINTEL cluster. The second was on how to prevent collecting data multiple times by keeping track of collected data.

Retrieving and writing data

Both the connection to Spark and InfluxDB needs to be set up once when the program starts and both need to be closed when the program ends. InfluxDB supports multiple methods of writing, but the default 'batching' method was chosen since this can write large amounts of

data points efficiently. Since the results from Spark may be very large, the system should iterate/stream over the resulting table and simultaneously write these rows (after conversion) to InfluxDB, without writing the whole table in memory.

Keeping track of collected data

In order to prevent the system from collecting data multiple times a method was needed to keep track of the data already collected. Two options were considered for this. The first option was to query InfluxDB to check what days were collected. The second option was to save data locally that contained this information.

For the first option the difficulties lay with InfluxDB. InfluxDB has no easy method to find for what timestamps data is collected. This can only be done by retrieving the whole series and writing code that loops over the data. This is inefficient; especially with large amounts of data. Alternatively, only the latest and earliest dates could be retrieved and used. However, this could cause gaps to become present in the data. This would then require someone to manually add this data.

For the second option data would be saved locally. The disadvantage is the increase in saved data. The advantage is that Flux knowledge is not required. It also allows for saving more data, including the gaps that could cause an issue.

Due to the simplicity and issues with the first solution the second solution was chosen.

Saved data

Due to the choice to save local data, choices needed to be made about what data was saved locally.

Since we needed information about the query to check if the query had changed, it was decided to copy all fields present for the measurement into the local data. When a change was then made to a measurement it could be detected.

For keeping track of what days were already collected there were several options.

The first was to save a start- and end-day, where data is known to be collected for all days in this range. The advantage is that it takes up little storage and is very easy to implement. The disadvantage is that if a query fails for a day the system might need to collect many days again. This is because the days before or after the failed day cannot be added to the range, since the system would then think the failed date was collected correctly.

The second option was to save a list of these ranges. This has the benefit over the first solution is that when a day fails it can continue while still saving what days are collected. The

disadvantage is that additional implementation is needed to calculate to what range a day needs to be added and to combine ranges together.

Due to implementation difficulties and the idea that failed queries would be rare it was decided to use the first option.

However, when there was not much time left, we discovered some issues with days missing in certain tables of OpenINTEL data. This created the issue that our assumption about frequency of failed queries being low was wrong. Since we were in the final stages of the project it was decided not to implement the second option but implement a quick fix. The fix was to also add the days that had failed in a list and change the day range as if the query had not failed. The advantage is that it was an easy addition to the system. But when a lot of days fail the data file increases in size.

Based on these decisions for each measurement name the information from the data configuration is saved. Additionally a start and end date for the measurements is saved combined with a list of all dates that have failed.

4.2.4. Historical and live data collection

Another choice that needed to be made was how to handle the situation where the system is still collecting and processing data when the system is started again. One of the situations in which this could happen would be if a new query is added and the system is started again, while another instance of the system was still processing. We decided that more recent data has a higher priority, since this is the data that is more likely to be viewed. For the given situation this would prioritize data collection for the new query. The following solutions were considered for this problem.

Running multiple instances

One solution that was considered was to let multiple instances of our system run at the same time. The problem with this is that information about collected data is saved locally and is changed by the system. This solution would create data races for this data unless a service was created to read or write data. This makes the system complex and requires a service to be running constantly. An advantage is that the system is expandable. If data needs to be collected faster one can just start more instances.

Running a service for collection

The next solution was to have a service running that collects the data. When the system is called to collect data, a script could add the data to a queue. This queue is then read by the service and data collected from the cluster. The problem with this solution is that the

SparkContext used for collecting and transforming the data needs to be closed when not used to save resources on the cluster. Furthermore, this solution has a similar issue as the first, since the script that adds data to the queue still needs to read the data. As a solution to this data race the service could handle access to the data. This however increases the complexity.

Signals
Another solution would be the use of signals. With this solution the system would first check if another instance is running. If there is no instance running the system can start collecting data. If there is an instance running the started system would send a signal to the running system to indicate it needs to check for new data. Since this solution is not multi-threaded or multi-processed it is less complex and there is no need to deal with concurrency issues. The use of signals does introduce some complexity.

Due to the complex nature and inherent issues of multi-threading/-processing we decided not to use the first or second solution, but the third solution using signals instead.

4.2.5. Public access

Grafana has the option to have anonymous user access. This allows everybody with access to the Uniform Resource Locator (URL) to see the dashboards. The disadvantage is that everybody gets access to all dashboards, folders, and data sources. Additionally, they can also make view calls to the Application Programming Interface (API) and can make arbitrary queries to any data source that is configured in Grafana [3]. Since InfluxDB uses tokens for access that can be limited to certain buckets and InfluxDB only contains time series data that are public, this is not an issue. The option to embed the dashboard would mostly avoid this issue. This is since the user is then only accessible by the client/admin. The website would need to be set up in such a way that the API is not exposed to people using this website. Another option would be the use of snapshots [4]. This however limits the capabilities of the dashboard (such as variables) and has significant overhead when multiple dashboards are needed. Due to the simplicity and limited disadvantages the choice was made to use Grafana directly as the website.

4.2.6. Security

InfluxDB runs in Docker containers that has the web port forwarded to the local host. HTTPS is not set-up, since this port is bound to only the hosts local loop-back address. Connecting to the host requires a Secure Shell (SSH) tunnel, which is secure.

Since both InfluxDB and Grafana run in Docker there is also no need to secure the connection between them since this is a separate network within Docker. Only when other Docker services would be connected to the same Docker network could this cause an issue.

Because Grafana is used as the front-end for our system it could benefit from an HTTPS connection. Since HTTPS requires setup with certificates it was decided not to set this up for the test system. Additionally, for the test system the Grafana Docker image's web interface was only connected to the local host the same way as InfluxDB.

Another issue both Grafana and InfluxDB have is that when metrics are enabled everyone can access them. These metrics show the performance of the InfluxDB and Grafana systems, it is undesirable to have these public. No method was found to make the metrics private, so it was decided to disable the metrics.

To only allow admins to manage the database and dashboard, some sort of authentication is needed. InfluxDB provides a simple username/password authentication whereas Grafana also provides other (third party) authentication methods. For Grafana to query the data from the database there needs to be a connection between these parts. To make this secure, an API token should be used, which needs to be stored in a safe place. Preferably the database should not allow any incoming connections apart from Grafana, which can be done by configuring a firewall.

For the metrics issue a reverse proxy could be used to prevent users access. This reverse proxy could then also be used for the HTTPS connection.

5. IMPLEMENTATION

In the following subsections the implementation, split into respective parts, are explained in more detail. A class diagram for the system was created and can be seen in the Appendix - Figure 3.

5.1. General system

We chose to make our system to be compatible with the Python version that was already installed on the given Virtual Machine (VM). This was Python version 3.8.

For checking that there was only one system running at a time we used a library that allowed us to use a lock-file. This library also saved the Process ID (PID) in this file. This allowed us to check if another system was running and if it was, use the PID to send a signal to this process.

5.2. Data retrieval

Data retrieval from the HDFS is handled by using PySpark [1] - a Python library - and thus indirectly using Apache Spark [9]. The data can be retrieved from individual warehouses, after which this retrieved data is saved as a persistent table. It is also possible to select data from multiple warehouses into the same table, as columns are similar.
The definition for these tables is done in the *table_config.ini* config file in the config directory. Here one can simply specify the warehouses and name for a proposed table. These tables can be retrieved the next time the system runs and are then able to be queried using (Spark)SQL [2].

5.3. Data processing

Data processing is done mostly using SparkSQL [2], using properly made SQL queries one can make sure that only the necessary data is retrieved. This data could then be transformed inside the queries themselves; using - for example - SQL case-statements. Once the data is transformed it is put into a PySpark DataFrame.

5.4. Writing data to InfluxDB

To write the retrieved and transformed data to InfluxDB the DataFrame is iterated over and each row is written with the given tags, as in the data_config.ini (ref: 8.1.3) and the rest are made values. For this to happen, each row is first converted to a Point object from InfluxDB.

Changing a query

As a query is changed in the data configuration, this change is also reflected in InfluxDB. Since InfluxDB overwrites the data fields with the same measurement name and timestamp; measurements, which queries were changed, can simply be overwritten.
A complication arises when tags and fields are changed. To give room for errors and keep the system simple it was decided that tags would not be removed by the system. Instead, a warning is given to the admin that the tags have changed. The admin can then remove the tags manually. Since the system does not remove tags, the dashboard also remains functional even if a tag was removed from a query. This gives the option to the admin to change the dashboard before removing the tag from InfluxDB.

5.5. Docker

The Docker containers of InfluxDB and Grafana were set up using Docker compose. This allows both the containers to be created and changed from one file. For both containers a Docker volume was specified. This allows the containers to save data on the hosts hard drive and make the data persistent. A network was also set up so that the containers could communicate with each other. This is needed for Grafana to retrieve the InfluxDB data. Both the web page front-ends were port forwarded so that one could view them from a browser. The last thing that was set up was an environment file for both InfluxDB and Grafana. These environment files contain settings for InfluxDB and Grafana.

5.6. Logging

The logging of warnings, errors or other information during run-time is done through the Python logging library, more specifically with the use of its TimedRotatingFileHandler Handler-class [6]. This handler regulates the logging so that one log per day is made in the *log directory*. In the log-config one is able to specify from which level logging should start [7] and how many old logs are to be kept.
Next to the TimedRotatingFileHandler, the SMTPHandler [8] is also used, this SMTPHandler sends an (optional) e-mail notification upon a severe error. The e-mail address for this notification is also changeable through the config. This config is mentioned in more detail later.

5.7. InfluxDB

InfluxDB uses buckets to hold measurements, which are the result of our queries, these buckets can be specified in the config files. This enables someone to direct where one's data is being saved and to segregate this data. This is what was done for the testing and live data. The testing data was simply saved to a *test* bucket, whilst the live data would be written to the *OpenINTEL* bucket.
Tokens have been generated with specific permissions; an admin token, a reader token, a testing token, and a writer token have been set-up. These tokens are used for the administrator or by Grafana, the testing functionality and the Python code respectively to portion off access to features and therefore make the system more secure. Additional tokens can be created and existing ones can be deactivated. The admin account is the account which is used to log-in to the web UI, where most of Influx' settings can be altered, buckets and measurements can be inspected and even alerts or tasks can be planned. A (simple) dashboard inside InfluxDB can also be found and used, but this functionality was not used for the system as it was redundant, since we are using Grafana.

5.8. Grafana

Since prespecified measurements are inserted into the InfluxDB, retrieving the data to be used for graphs mainly consists of querying these. This data is retrieved for the user selected time frame and could be filtered on specific values. These are filters such as one for query-type AAAA, when attempting to create an only AAAA graph, when using a measurement that contains more than just the AAAA query type.
After filtering, other functionality, such as grouping and taking sums might be used to get a complete value over the interval, rather than a time bound series.

Grafana also offers functionality to rename database types. For example, something that looked like *{count "query_type"="AAAA"}* could be transformed into *"Count AAAA"* by using this built in functionality. Next to this transformation, based on the graph there are also numerous other options - such as 'limit' - which is used in some cases to limit the amount of data, to - for example - not over-flood a pie chart.
A global dashboard filter is also used, namely the TLD filter to specify one or more specific TLDs in the graphs, if one for example would want to look at only data from the *.org* TLD this functionality could be used. To support this global variable, for each graph that would need to support this, its measurement requires a column for this variable to be looked-up in. For example, a Time To Live (TTL) per query-type measurement needs to have a column designated for the TLD, if one would want to filter this.

Keeping possible colours to blue and purple as much as possible, along with a white background we try and incorporate as much of the OpenINTEL website style into the dashboard. This could not be done everywhere, as certain graphs would become much more difficult to comprehend using a limited colour spectrum.

6. TESTING

In order to make sure that the system works as expected, multiple tests were be put in place and/or conducted. For the separate parts of the system unit test are used where possible. A system test was created to test if the whole system works.

6.1. Unit Tests

These tests make sure that parts of the system work. These tests do not actually use Spark or the InfluxDB, these are patched to see if the correct data was sent to these external parts. About 80% of the targeted files are covered by these tests.

6.2. System Tests

Even when all parts of the system on their own work, it does not mean the whole system works correctly, for this a system test is performed. Before this test starts, any previously generated data is removed. This test then (dynamically) generates multiple sources (replacing the HDFS) before running the program with predefined configuration files. After the program has ran, the expected data is written to InfluxDB, under another measurement name. These measurements are then compared, if indeed the correct data has been written using the InfluxDB testing library [5]. The check if the data is also correctly shown in Grafana, must be done manually.

7. Performance

System performance was examined in multiple locations using the relatively small size OpenCC data. Both querying, transforming and writing data; as well as the dashboard performance have been looked at more closely. These, along with their examinations and possible results, are described further below.

For the tests below the system was run on a VM with 16GB RAM and 8 CPU cores. The VM runs Ubuntu 18.04 LTS.

The performance tests have been run with the queries found in C.1. These queries were performed on the `opencc` data from OpenINTEL. This table is relatively small compared to the total OpenINTEL data set.

7.1. Querying, transforming, and writing data

To better examine the performance of certain measurements with respect to how their data is queried from the HDFS, transformed and written to the InfluxDB we have gauged their speed. For this we collected the time it took to run different parts of the code and how often this code was ran. The running times are collected for the collection of data, adding the data to the queue, running the query and writing the data to InfluxDB. For the measurements of querying, transforming and writing the data, the times were collected for each measurement separately to see the effects of the query.

In table 1 the running time of the different components can be seen. The average is taken based on the number of queries that have been run. Since adding data to the queue is independent on how many queries are ran, the average is not calculated for this measurement.

	Measurement 1 (12 queries)		Measurement 2 (367 queries)	
	Total	Avg. per query	Total	Avg. per query
Collection	11618.45 (193.6)	968.20 (16.1)	75883.22 (21 hours)	206.77 (3.5)
Adding to queue	0.62		0.48	
Querying and transforming	11610.50 (193.5)	967.54 (16.1)	75675.93 (21 hours)	206.20 (3.4)
Writing to InfluxDB	7.31	0.61	206.13 (3.4)	0.56

Table 1: *System time performance: time in seconds (and minutes) for different parts of the system*

As can be seen from the results the time it takes for the system to collect data mostly depends on the query through Spark. It can also be seen that this can fluctuate based on the time that the query is ran.

Table 2 shows the time different queries took. For each measurement its related SQL query can be found in appendix section C.1. In this data config the name inside the square brackets is the name of the measurement.

	Measurement 2 (367 queries)		
	Query and transform	Writing to InfluxDB	Number of writes to InfluxDB per query
Query_type	314.77	0.21	511.2
country_counts	88.75	0.22	706.9
A_vs_AAAA	92.58	0.08	187.8
ttl_A_AAAA	68.39	0.05	85.5
ttl_MX_NS	67.26	0.03	44.2
timestamp_interval	309.75	0.08	159.3
country_counts_union	128.05	1.89	8994.7
country_counts_opencc_alexa	126.21	2.15	8509.0
total_datapoints_and_domains_opencc_alexa	653.30	0.32	810.9

Table 2: *Query time performance: average time in seconds for each query and the number of writes per query*

The results show that again performing the query is the bottleneck of the system. Additionally it can be seen that the time it takes to query can differ a large amount. With the total_datapoints_and_domains_opencc_alexa taking almost ten times as long as the TTL queries. It can also be seen that some queries cause more writes to InfluxDB. These queries are also the queries where writing takes longer. Whether this is due to more data being sent or some other mechanism could not be determined.

From these measurements it can be seen that the system can perform all queries well within a day. It does need to be noted that the measurements were run on smaller tables from the OpenINTEL data set. With larger tables used there might be a decrease in performance.

7.2. Dashboard performance

The dashboard performance has been tested with the use of Selenium after having inserted a six months worth of data into the dashboard. So that we can compare this to our requirement to serve the end user with (adjusted) graphs of at most a year's interval within 10 seconds. Using Selenium we measured 5 page load times per dashboard view using a single TLD (.se / Sweden's ccTLD), a 100 random TLDs and all TLDs in the TLD selector, over the course of 6 months.
The results of these measurements are contained in the table below.

	Time 1	Time 2	Time 3	Time 4	Time 5	Avg. Time
Default Dashboard TLD = .se	6.10	4.91	4.84	4.89	4.92	**5.13**
Time Dashboard TLD = .se	2.59	2.66	2.61	2.61	2.44	**2.58**
Default Dashboard TLD = 100 random	11.64	10.32	10.38	10.20	10.45	**10.60**
Time Dashboard TLD = 100 random	7.56	7.16	6.99	7.21	7.28	**7.24**
Default Dashboard TLD = ALL	4.71	3.82	3.82	4.25	4.14	**4.15**
Time Dashboard TLD = ALL	3.02	2.45	2.91	2.60	2.50	**2.70**

Table 3: *Dashboard Time Performance: Seconds for each page load depending on TLD selector and dashboard for a 6 month time range*

Initially, the loading times were much larger than the ones displayed, this was mainly due to the TLD selector. Loading times for both dashboards were around two minutes for the regular one; and fifty seconds for the time dashboard. Due to improvements mainly consisting of checking the size of the TLD selector and disregarding it whenever more than 999 values are selected, outside of the query, loading times were brought down to around what it is now. This means whenever a fairly high number of TLDs (lower than 999) are selected the loading times are quite slow. When a lower number of TLDs (less than 100) are selected; one can however expect quicker loading times, as is shown in the table.

7.3. Conclusion

The performance of querying the data from HDFS is fast enough to query all data for a day within one day. Additional queries and larger tables could decrease the performance.

System performance was only tested by means of time to complete. Additional info such as CPU usage and memory usage were not taken into account within these tests. System performance was manually observed during operation using htop. Here the system uses at most 200% CPU during the collection of data. When loading the dashboard, InfluxDB seems to use all available CPU resources to query the data. As for memory usage, querying from HDFS with spark uses the most memory with around four GB of memory usage. InfluxDB uses less than one GB and Grafana uses closer to 100 MB.

Since CPU and memory usage were not tested and the effect of multiple requests to the dashboard having an unknown effect on loading times, hardware recommendations are based on best guesses.

	Minimum	Recommended
CPU	4 cores	8 cores or higher
Memory	8GB	8GB

Table 4: *Hardware requirements*

The dashboard loading times were quite lengthy, but due to query improvements we managed to get these time below the requirement (ref: 2.2.2.1) for the - in our eyes - viable use cases. These include selecting a small number of TLDs and selecting 'All' TLDs. Any selection of over a hundred and less than 'All' TLDs did not seem to be a likely choice as most of the graphs would be come badly readable with that much data. Therefore we think this check is sufficient.

It should also be mentioned that not all performance issues can be properly addressed, some issues are inherent to Grafana and InfluxDB. Such as having to use 999 for the TLD check as a dirty fix, as we were not able to query the exact count due to Grafana restrictions.

8. System Usage

Below we describe certain system aspects, along with additional items that increase ease of use for the system for future operation.

8.1. Config files

To make future adjustments and additions less complicated, three config files are set up: the main-, data- and table-config. Respectively containing adjustable variables, executable SQL queries and warehouse to table definitions. Further information is given in each respective subsection below.

8.1.1. Main Config

The config.ini is the main configuration file. In here tokens, logging levels and other Boolean variables are stored. In this file the client should be able to change a step in the overall implemented process by changing one or more config variables. Further information on how to do this, is explained in the manual.

8.1.2. Table Config

The table_config.ini is used to define what data sources are used and which tables can be used for the queries specified in the data config. Here - for example - one can for example define that the table 'union' should consist of both 'Alexa' and 'OpenCC' data, which are both two different HDFS warehouses. Since the columns for the warehouses are all similar, these can be used both in union using this method or by simply taking a SQL-union of two different tables, which have been defined in this config file.

8.1.3. Data Config

The data_config.ini is the location for all executable queries are saved, these can be added in the Spark-conform-SQL-format [2]. They are executed and written to either the default bucket as given in config.ini or another bucket if present in the possible keys for a specific query. For more information OpenINTEL can reference the manual.

8.2. Manual

To provide the client with instructions on how to expand the current implementation a manual is provided, this manual describes the process, how to set up the system along with required features, how to run the program and how to add or alter an existing measurement and/or graph.

8.3. Provided Jupyter Notebook Python code

Next to the manual, we provide the client with a Jupyter Notebook to test prospective queries. The notebook uses parts of the code of the system itself. The notebook is saved in the repository the client has access to, with additional annotations as to make the process assuredly more understandable.
By using this code the client should quickly be able to determine, if a created query gives the wanted result, so that it can be used in the system itself.

9. Discussion & Conclusion

9.1. Conclusion

In this subsection we consider the fulfilment of the requirements as presented in section 2 at the start of this document.

9.1.1. Must

- ✓ The system back-end must require authentication
- ∼ The system must process newly added data after 3 days
- ✓ The system must collect the data from the HDFS
- ✓ The system must be structured so that dashboard can be expanded by a few simple interactions
- ✓ The system must be able to store the time series data acquired from the OpenINTEL cluster

Due to issues with access to the data of OpenINTEL there were days that could not be collected after August. This meant that it was not possible to add data 3 days after it was available. Once this issue is solved the system is still able to add this data.

9.1.2. Should

- ∼ The dashboard should have quick loading times
- ✓ The system should be able to handle future data
- ✓ The system should be documented, so that an outsider could expand it
- ✓ The dashboard should handle different kind of data
- ✓ The dashboard should be able to show different kind of visualizations
- ✓ The system should be able to process historic data
- ✓ The database and front-end should run in a Docker container
- ✓ The system should log its actions
- ∼ The dashboard should allow filtering
- ✓ The system should handle SQL queries with admin defined table names

The dashboard loading times - as stated in 7.2 - are quite fast, but depending on the TLD selection; not perfect. The speed for a full year's worth of data was not examined exactly - as we did not have a full years worth of data yet and the used dataset mainly consists of OpenCC data.
The problems with loading times are partially to blame on the TLD selector and an under-

estimation as to how much computing time adding TLDs to the measurements would add. Optimizations have however already been made and could possibly be extended.
Filtering is implemented as far as the time and TLD are concerned, however for each filter that one would need to add one would need to append the data to filter on specifically to the measurements, as was done with the TLD.

9.1.3. Could

- ✗ The rest of the system could run in a Docker container
- ∼ The system could send a notification when an error occurs
- ∼ The dashboard could be styled like the OpenINTEL website

Due to time constraints and difficulties related to Spark, our system was not made into a Docker container.
There are some limitations in the email handler and its security, therefore we think this requirement could have been better implemented.
Due to limitations of Grafana with respect to themes; the dashboard is not styled completely like the OpenINTEL website. It was set to a light theme and where possible graphs were given the colors of OpenINTEL.

9.2. Discussion

Reflecting back on the project, the work we have done and the approach we have taken to get the work done, we are satisfied with the end result. Almost all of the requirements are implemented as to the clients wishes and we have learned a lot in the process. Communication, both within the project group as with the client, was good. Planned daily meetings on Teams plus some individual work afterwards worked greatly to get everyone on the same page and to get our work done in order.
The use of GitLab was helpful, as there we were able to track issues and problems properly, although in the last couple of weeks the usage of GitLab declined.

After the halfway point the planning was not checked anymore. Since most of the project was on schedule and the system was mostly complete it did not cause problems. When things would have gone wrong, not checking the planning could have been an issue.

9.3. Individual Contributions

For the contributions we will consider every aspect in the design, development, testing and documenting process. For these aspects it will be denoted who worked on said. Next to these contributions, which we are mainly done outside of the morning group meetings, it should also be noted that during these meetings other work was done in unison.

	Chris	Bart	Ruben
First Influx & Grafana Demo	X		
First ElasticSearch & Kibana Demo		X	
Overleaf & Document layout setup			X
Second Influx & Grafana Demo		X	
Second ElasticSearch & Kibana Demo	X		
First TimeScaleDB & Grafana Demo			X
Initial Grafana & InfluxDB VM set-up			X
Initial GitLab set-up	X		
GitLab pipeline		X	
Data processing exploration	X		
Grafana & InfluxDB docker set-up		X	
Making queries in Grafana			X
SQL queries			X
Data processing classes	X		
Data collection classes		X	
Logging	X		X
Configuration for the program		X	
Processing meetings into document			X
Contact with client/supervisor via mail			X
Making of unit tests	X	X	
System test	X		
Performance tests	X	X	X
Documentation	X	X	X
Document writing	X	X	X
Presentation(s)	X	X	X
Poster	X	X	X

9.4. Future Implementation

Below we denote and discuss some possible expansions or changes to the delivered system as to give OpenINTEL some inspiration to take this to a next level.

Adding support for data sources other than OpenINTEL.
By loading data from elsewhere than the OpenINTEL HDFS data, other sources could be added to the dashboard. For example, data from SIDN could be linked, so that this could be used to make comparisons.

Adding support for multiple time ranges in the local data to prevent collecting days multiple times.
Currently the system only saves a start and end date along with a list of failed dates in the locally saved data. This is not the most optimal way regarding storage space. To improve this the system could be changed to store multiple date ranges of dates that have been collected.

More efficient use of dataframes
Currently dataframes are generated once every query, it would be more efficient if certain dataframes, that are used for multiple queries are generated once outside of the query loop.

Add TLD column to dataframe, outside the query loop
The above dataframe generation includes finding the TLD for each response name. The query process could be optimized if this was already done outside of the query (loop). This TLD transformation could then also use caching since most domains are also present in data of other days.

Move system to Docker
To make the system easier to move and to reduce system setup, the system could be moved to a Docker container.

A. Definition

A.1. Acronyms

API Application Programming Interface. 18, 19

ccTLD Country Code Top-Level Domain. 3, 26

DACS Design and Analysis of Communication Systems. 3

DNS Domain Name System. 3, 4

GUI Graphical User Interface. 11, 14

HDFS Hadoop Distributed File System. 6, 8, 13, 20, 24, 25, 27, 29, 31, 34

HTTPS Hypertext Transfer Protocol Secure. 7, 18, 19

INI Initialization File. 15

JSON JavaScript Object Notation. 15

PID Process ID. 20

RAM Random-Access Memory. 9, 25

SQL Structured Query Language. 7–10, 13, 20, 26, 29, 31

SSH Secure Shell. 18

TLD Top-Level Domain. 3, 23, 26, 27

TTL Time To Live. 23, 26

UI User Interface. 9, 10, 12, 22

URL Uniform Resource Locator. 18

VM Virtual Machine. 20, 25

A.2. Glossary

Docker A set of platforms as a service product that use OS-level virtualization to deliver software in packages called containers. 9, 12, 18, 19, 21, 31, 32, 34

Hadoop The Apache Hadoop software library is a framework that allows for the distributed processing of large data sets across clusters of computers. 10, 15

HBase Apache HBase™ is the Hadoop database, a distributed, scalable, big data store. 10

Jupyter Notebook The Jupyter Notebook is an open-source web application that allows you to create and share documents that contain live Python code, equations, visualizations and narrative text. 8, 30

Time Series Daemon Service that runs on the machine which is responsible for interacting with HBase and store/retrieve data. 10

Time Series Database A software system that is optimized for storing and serving time series through associated pairs of time(s) and value(s). 8, 11, 13

B. Bibliography

References

[1] PySpark documentation.
https://spark.apache.org/docs/latest/api/python

[2] Spark SQL Syntax documentation
https://spark.apache.org/docs/latest/sql-ref-syntax.html

[3] Security implications of enabling anonymous access in Grafana
https://grafana.com/docs/grafana/latest/administration/security/#implications-of-enabling-anonymous-access-to-dashboards

[4] Snapshots in Grafana
https://grafana.com/docs/grafana/latest/sharing

[5] Testing in InfluxDB
https://docs.influxdata.com/influxdb/latest/reference/flux/stdlib/testing

[6] TimedRotatingFileHandler - The handler-class used for logging per day.
https://docs.python.org/3/library/logging.handlers.html#timedrotatingfilehandler

[7] Python Logging library logging level
https://docs.python.org/3/library/logging.html#levels

[8] SMTPHandler - The handler-class used for sending severe error notifications per mail.
https://docs.python.org/3/library/logging.handlers.html#smtphandler

[9] Apache Spark™ is a multi-language engine for executing data engineering, data science, and machine learning on single-node machines or clusters.
https://spark.apache.org

C. Appendix

C.1. Data config

```
[Query_type]
sql_query : SELECT query_type,
        regexp_extract(lower(response_name) , '(?:\.)([^.]+)(?:\.)$') as
        ↪ tld,
        COUNT(query_type) AS total
        FROM opencc
        GROUP BY query_type,tld
tag_names: query_type,tld

[country_counts]
sql_query : SELECT country, regexp_extract(lower(response_name) ,
↪ '(?:\.)([^.]+)(?:\.)$') as tld, count(country) as count
            FROM opencc
            WHERE query_type in ('A', 'AAAA') AND response_type in ('A',
            ↪ 'AAAA')
            GROUP BY country,tld
            ORDER BY count DESC
tag_names : country,tld

[A_vs_AAAA]
sql_query : SELECT query_type, regexp_extract(lower(response_name) ,
↪ '(?:\.)([^.]+)(?:\.)$') as tld, count(*) as total
        FROM opencc
        WHERE query_type in ('A', 'AAAA') AND response_type in ('A',
        ↪ 'AAAA')
        GROUP BY query_type, tld
        ORDER BY query_type DESC, total DESC
tag_names: query_type,tld

[ttl_A_AAAA]
sql_query : SELECT rrsig_type_covered,
        (CASE WHEN rrsig_original_ttl <= 300 THEN 1
        WHEN rrsig_original_ttl > 300 AND rrsig_original_ttl <= 3600 THEN
        ↪ 2
```

```
        WHEN rrsig_original_ttl > 3600 AND rrsig_original_ttl <= 86400
        ↪ THEN 3
        ELSE 4 END) as ttl_bracket,
        regexp_extract(lower(response_name) , '(?:\.)([^.]+)(?:\.)$') as
        ↪ tld,
        count(*) as count
        FROM opencc
        WHERE rrsig_type_covered in ('A','AAAA')
        AND rlike(response_name, '^[a-zA-Z0-9\\-]+\\.[a-z]+\\.$')
        GROUP BY rrsig_type_covered, ttl_bracket,tld
tag_names: rrsig_type_covered,ttl_bracket,tld

[ttl_MX_NS]
sql_query : SELECT rrsig_type_covered,
        (CASE WHEN rrsig_original_ttl <= 300 THEN 1
        WHEN rrsig_original_ttl > 300 AND rrsig_original_ttl <= 3600 THEN
        ↪ 2
        WHEN rrsig_original_ttl > 3600 AND rrsig_original_ttl <= 86400
        ↪ THEN 3
        ELSE 4 END) as ttl_bracket,
        regexp_extract(lower(response_name) , '(?:\.)([^.]+)(?:\.)$') as
        ↪ tld,
        count(*) as count
        FROM opencc
        WHERE rrsig_type_covered in ('NS','MX')
        AND rlike(response_name, '^[a-zA-Z0-9\\-]+\\.[a-z]+\\.$')
        GROUP BY rrsig_type_covered, ttl_bracket,tld
tag_names: rrsig_type_covered,ttl_bracket,tld

[timestamp_interval]
sql_query : SELECT regexp_extract(lower(response_name) ,
↪ '(?:\.)([^.]+)(?:\.)$') as tld, MIN(timestamp) as minTimestamp,
↪ MAX(timestamp) as maxTimestamp,
            (MAX(timestamp) -MIN(timestamp)) as timestampInterval FROM
            ↪ opencc t1
            GROUP BY tld
tag_names: tld

[country_counts_union]
sql_query : SELECT 'opencc' as source, country,
```

```
                                regexp_extract(lower(response_name) ,
                                ↪  '(?:\.)([^.]+)(?:\.)$') as tld,
                                count(country) as count
                                FROM opencc
                                WHERE query_type in ('A', 'AAAA') AND
                                ↪  response_type in ('A', 'AAAA')
                                GROUP BY country,tld
                        UNION
                        SELECT 'alexa' as source, country,
                                regexp_extract(lower(response_name) ,
                                ↪  '(?:\.)([^.]+)(?:\.)$') as tld,
                                count(country) as count
                                FROM alexa
                                WHERE query_type in ('A', 'AAAA') AND
                                ↪  response_type in ('A', 'AAAA')
                                GROUP BY country,tld
                        ORDER BY count DESC
tag_names: source,country,tld

[country_counts_opencc_alexa]
sql_query : SELECT 'union' as source, country,
                                regexp_extract(lower(response_name) ,
                                ↪  '(?:\.)([^.]+)(?:\.)$') as tld,
                                count(country) as count
                                FROM opencc_alexa
                                WHERE query_type in ('A', 'AAAA') AND
                                ↪  response_type in ('A', 'AAAA')
                                GROUP BY country,tld
tag_names: source,country,tld

[total_datapoints_and_domains_opencc_alexa]
sql_query : SELECT regexp_extract(lower(response_name),
↪  '(?:\.)([^.]+)(?:\.)$') as tld, COUNT(DISTINCT response_name) as
↪  domains, count(*) as points
                FROM opencc_alexa
                WHERE response_name <> '.'
                GROUP BY tld
tag_names: tld
```

C.2. Class diagram

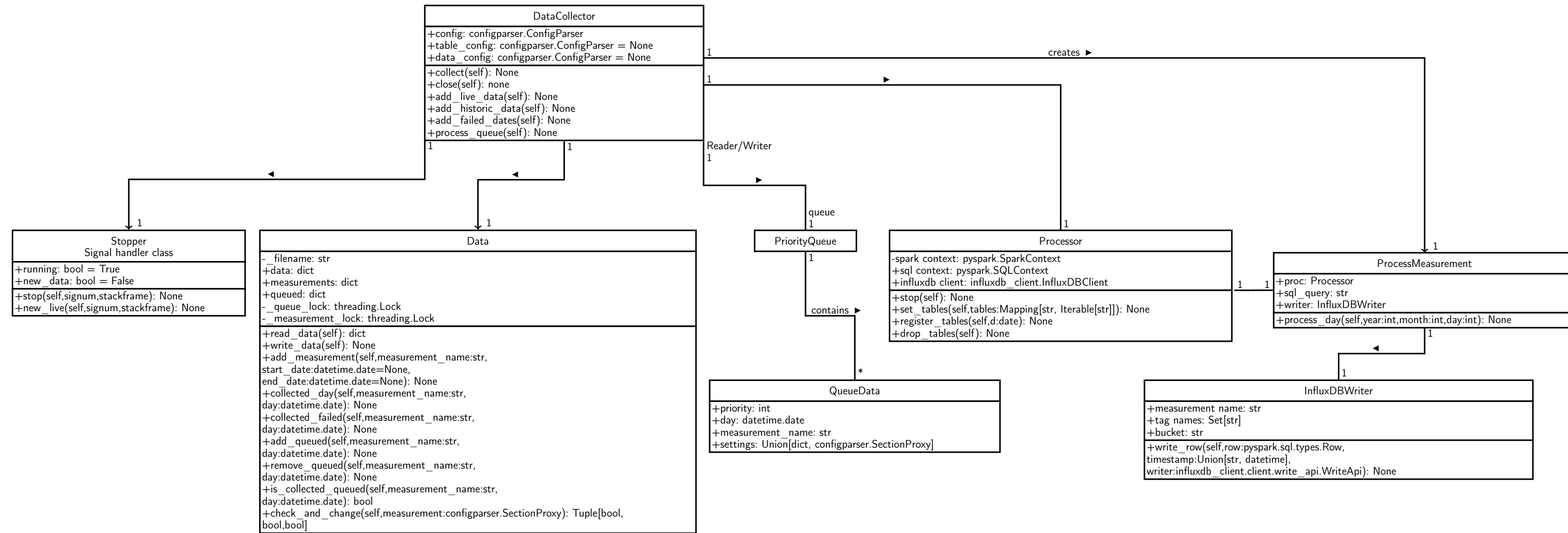

Figure 3: *Class diagram for the implementation*

C.3. Meetings

Underneath we give summaries of our meetings with Raffaele Sommese, both our client and supervisor, who works with DACS and OpenINTEL. These meetings either happened in person or using video conferencing software.

First Meeting - September 8th 2021 15:00

Our first meeting with Raffaele. We had agreed on the meeting during our initial mail contact and had already been given some information resources to study beforehand. After studying the information provided we managed to make ourselves known with the broader general idea of what we were meant to implement.

Naturally we did have some questions, these we noted in a collaborative online document to be asked during the meeting. These are summarized below; including their answers.

What is the audience (e.g., scientists, interested people, people with little knowledge)?
Everything will be public (in general), it would most probably still be people interested in the data (researchers/journalist/companies).

Dashboard available for anyone? Authorisation, authentication?
No authorisation/authentication required, dashboard should be available for everyone.

From what place does the application gets the data: Kafka (live data from measurements) and/or HDFS (long term storage)? Spark (data analysis)?
It was suggested we should retrieve the data from Spark.

Does this data need to be altered/processed? Since it's a lot of data points. Do we need to store any (additional) data?
Just the day is necessary; we wouldn't need to keep track of the additional time the different worker nodes get information.

How 'live' should the data be? (e.g., direct from source and/or without refreshing page?)
Once a day we should insert that day's retrieved data. The design should sustain the load of data. We should pay special attention to how we transport the data from point A to B (pipelines).

Will we be provided with a development/testing/deployment server? (Or what kind of server will the application eventually run on, e.g. Linux?)

We will be given access to a Virtual Machine to implement on and a portion of the data to use before next week. We should start with a larger ccTLD to begin with.

Predefined graphs, or should the user have lots of flexibility/edit-ability (e.g. filtering/sorting)?
We were given an example website, Raffaele suggested we should stick to this example for now.

Should it be possible to add graphs after the application is 'done'?
Yes, this would be very helpful; Raffaele would not like to spend a lot of time digging through our code and rather have it be provided in an instruction manual. Everything should be kept well documented.

Preferences for programming languages/frameworks? Importance of quality of code, documentation, modularity, colour scheme, language (Dutch?), exportability of graphs, etc.
None, we're free to use what we like, as long as we import and display their data. The dashboard should however have somewhat the same style as their current website.

Further points:

- Important to show how we collaborated and planned things.
- We have to give good (might be personal) basis for making certain design choices.

Second Meeting - September 13th 2021 15:00

Our second meeting we planned quite quickly after the first, this was because we wanted to get the project moving as quick as possible. The interval gave us enough time to look at some possibilities, deliver a project proposal draft for feedback and have follow-up questions ready for Mr. Sommese. These questions mainly arose out of the demo's we had created/done. This meeting was done through Zoom. These questions are listed down along with some additional feedback on the design proposal.

VM / access to data?
It was mentioned that Mr. Sommese would try to get this set up before Wednesday, we received an e-mail with details that same evening.

What kind of environment (OS) will the final product have to run on?
VM is on an Ubuntu 18.04 machine

When adding a new graph does it also need to add historical data? What data is more important old, or new? Or does it need to be select-able?
Also historical, although it might not have to contain all data.

Is there a preference of SQL as query language versus Flux (InfluxDB's language)?
Flux would be okay with proper documentation.

Is there a need or requirement for back-ups?
We may have back-ups.

Is there a need or requirement for using docker?
We should deploy with docker if possible.

Additional feedback

- We need to add more detail about requirements, along with additional explanation about system requirements.
- We need to add an introduction to the proposal.
- More testing of the technologies
- VM we will get access to will have some data, connection to cluster and an introductory Jupyter Notebook installed.
- In terms of planning more parts could be implemented in parallel.

Third Meeting - September 20th 2021 15:00

Before this meeting Ruben, on request of Mr Sommese had signed an NDA so that we could get access to the cluster. In this meeting we discussed in which way we should query data from this cluster and talked through requirements. We were also told that it would be fine if we supplied - for example - a year of historical data, just so that the graphs would not be empty.

Requirements feedback

- We should change our references from Spark to HDFS, Spark is a program - HDFS the actual storage.
- Specify time series (not just processed data) in the figures.
- Don't specify avg. 10 seconds load time, but present worst and best case scenarios.

  Note: Specify for example '10 seconds when the only request for one year of data'.
- Use/specify different graph types i.e. line, bar, pie, static.
- Required to fill historical data when -for example - adding a new column.

Next to these requirements we received some feedback on the overall design proposal.
Mr Sommese told us that docker was more required than we had previously thought, this threw kind of a spanner in our works w.r.t. the planning and the environment we had set up so far on the VM.
Furthermore, we should look further into Grafana themes or customization options and not use snapshots as they might have problems the problems of being static and badly scale-able.
As for our data retrieval and transformation, we should do most of the transformation - if not all - using Spark.
Add usability testing, possibly get feedback from outsiders. Furthermore, after asking for more kinds of examples as to what queries we should build we were referred to the SIDN graph website.
As for accessing Spark we should use specify 'NS' queries for all responsive domains record and 'SOA' for all records, responsive or unresponsive. As source we should use 'opencc' for time series and not 'alexa'. Retrieve the data from spark per day, you can loop over the retrieval days, but this would be quickest and then try that for one month. Don't load data from the past three days - as that might not be fully stored on the cluster yet.

Fourth Meeting - September 27th 2021 15:00

Over the last week, where we did further implementation on the pipeline and set up some PySpark queries to query specific data we wanted to showcase, some questions arose. These questions were asked during this meeting. Next to this, before this meeting, we had set up the pipeline, from HDFS using PySpark to InfluxDB and with InfluxDB coupled with Grafana already we were able to execute queries and create graphs.
Underneath we will list the questions we asked and the answers we received.

In terms of the dashboard, would you prefer using Grafana to export the whole dashboard or panels.
Exporting the full dashboard is fine.

Where to find certain response codes (used for queries) and their definition?
We were given a short explanation and Mr Sommese linked us two websites we could use for further questions:
`https://datatracker.ietf.org/doc/html/rfc2929#page-4` and
`https://openintel.nl/background/dictionary`

Are there limits to data collection range?
Just one query per day is fine (nothing else should be necessary).

Next to this Mr Sommese shared with us some example queries SIDN had used. We were informed we should transform all of the data to lowercase, since domains would not all be

lower case, for example Google domains could both have 'GOOGLE.COM' as 'google.com' which means would be counting Google domains twice, if we don't transform to lower case.
We were also informed that Mr Sommese would like to have a dropdown at the top of the dashboard, so users could select specific TLDs and for visibility reasons some graphs should be limited to a top 10 (possibly with an 11th for "Other").
A logging and error system should also be set up, preferably with a notification on fail - possibly through e-mail.

Fifth Meeting - October 1st 2021 11:00

Last meeting we had decided this meeting would have to be planned in the same week, since Mr Sommese would be more or less absent next week.
We showed Mr Sommese the current state of the dashboard and its graphs after which we received some more comments. These comments included proposed changes to the graphs as well as directions about what we should look at in the future. Same as in previous meetings we asked some questions, these will be listed below.

In our final application, can we keep our SparkContext open or would we have to close it?
You should close it whenever you're done.

Should the end user see a Grafana interface or HTML page with IFrame or the dashboard itself?
The Grafana dashboard is completely fine.

Furthermore, we noted that our Spark Access was denied from the fifth of September onward, after which Mr Sommese told us to send and e-mail so that he could forward this to his colleague, which we did.
Also on the dashboard should contain a link to the OpenINTEL website.

Sixth Meeting - October 11th 2021 15:00

Next to Mr Sommese, Mattijs Jonker was also present for the second part of this meeting. He manages the development on the Big Data side of OpenINTEL. So his potential feedback would be very much relevant to our project of putting this big data into an oversee-able scope.
During the meeting we first talked about some problems we encountered, Such as that we hadn't really received access to a part of the OpenCC data yet (which we use for testing).
Next to this we showed Mr Sommese how the configuration files are set up and would manage the overall process. This was later shown again when Mr Jonker was present. We received

positive feedback on both this set-up through configuration files as well the set-up of the dashboard and queries.

Following a short demonstration about the pipeline and workings of the dashboard we asked some questions, these will be listed below.

Would you expect more than 100 viewers at the same time, as this is the default limit for Grafana; otherwise we would need to set this higher?
You shouldn't worry about it too much, if scalability ever becomes an issue than we will look into it ourselves.

Metrics for InfluxDB & Grafana were available, we have disabled them would you like to be able to see them?
Only if this can be limited to an admin account, otherwise disable the metrics.

What should we do when a query is altered, should past measurements be removed automatically?
Overwrite measurement if a query is altered.

What is a query is removed from the config?
This question wasn't necessarily asked but both Mr Sommese as Mr Jonker mentioned the ability to drop a measurement manually.

For an SMTP handler to send error per e-mail, is there an SMTP server available or would we set up our own?
We were advised to set up our own SMTP, as Teehuis regulations did not necessarily allow for e-mails to be sent from there.

After this we received some additional feedback and directions. These included that we should include a way to specify different tables (HDFS storage locations) to be queried, so that these could be joined on each other. Furthermore, we should create a demo (perhaps in the manual) of how to go from config till live graph. We could also attempt to stress test the dashboard, although this would have low priority and measurements such as time per different Spark query per day would be more beneficial to show. The total per country graphs should be transformed to average per country. As well as a query where MX servers were located.
Perhaps in the design report we could also describe next steps or improvements to this system, for after we have delivered it.

Seventh Meeting - October 18th 2021 15:00

Since we had managed to plan our demo presentation last week, for next week Wednesday (the 27th), some questions had arisen with regards to this meeting.

How much time is available for us to demo?
Time has not necessarily been reserved, stick to about 10 to 15 minutes, 10 minutes for presentation 5 minutes for questions.

Where would the presentation be?
Not yet sure if all physical, if physical we can meet Mr Sommese 10 minutes beforehand at his office in Zilverling.

Who will be present, is there enough knowledge about OpenINTEL et cetera to not have to go into much explanation on this part?
Enough knowledge about OpenINTEL, do a short into about your project. Try to stick to a more global level throughout.

Next to the questions we mentioned that we were still getting permission errors on some time ranges, Mr Sommese told us to simply disregard them and plot data from some time before, as this might have to do with possibly bad HDFS disks.
Furthermore, Mr Sommese mentioned that we shouldn't spend too much time trying to fit the Python solution into a docker container, considering the dependencies, as long as we mentioned the set-up in the manual it should be fine.
Next to this some graphs on the dashboard were also still in need of change. For example the Y-axis in some cases could be set to hours instead of seconds and instead of query_type the TLD could be used for the timestamp interval graph.
Our next meeting would be coming Monday, beforehand we should prepare the demo presentation for Wednesday, so Mr Sommese could review it.

Eighth Meeting - October 26th 2021 15:00

Originally this meeting was planned on Monday like the others, but rescheduled to Tuesday. During this meeting we asked a few questions relating the presentation we would be giving the next day - on Wednesday. To the whole DACS group, the presentation would be held before their own group meeting and would. This meeting would be held through Zoom, for which we received a link as well. Furthermore, we noted to Mr Sommese that a specific path for OpenCC data (21/05/2021) did not work.

Ninth Meeting - November 08th 2021 15:00

The week before, due to little changes and no questions we purposely requested for the meeting to be moved to the 8th of November. The week before we had worked on this report and afterwards sent it as a draft to Mr Sommese. During the meeting we asked some final questions regarding the hand-in and last week as well as received some feedback on the draft we had previously sent.

The questions we asked, including their answers; or answers that were given not specifically to answer a question:
What time should we deliver our work coming Friday?
Any time on Friday is fine.

Do we need to black out any - possibly confidential - information in our report?
No, you can leave the report as it is.

How should we hand everything in?
We should give Mr Sommese access to our repository, we can add our documentation/manual as well as the presentation slides to the repository as well. The report we can deliver by mail.

As for the feedback on the report:

- The introduction/motivation can be extended with that the dashboard can contentiously provide insight into the status of the DNS as well as provide help in assessing research.
- It is currently not clear how we keep track of collected data, how the process of storing dates works and what happens when days fail. We should extend and better define this section.
- Security: HTTPS can be remedied by reverse proxy with apache/nginx.
- In dashboard performance we could give machine info and approximate hardware requirements (minimal & recommended)
- In the dashboard performance section we could include a part on concurrent requests.