Silesian University of Technology

Faculty of Automatic Control, Electronics and Computer Science

Control, Electronic and Infromation Engineering

Engineering thesis

Containerization of data warehouse creation and management processes

Author: Karol Latos

Supervisor: dr inż. Anna Gorawska

Gliwice, February 2022

Contents

Chapter 1

Introduction

The subject of this engineering thesis is a standalone data warehouse built using multiple Docker containers. The thesis describes how the data is gathered, processed and analyzed, and what components make up each container, as well as their mutual coordination. Additionally, practical results of the project are presented in form of answers to business questions, together with steps taken to achieve performance optimization and code robustness.

The basic assumptions of the system are the following:

1. The application is portable, standalone and platform-independent.
2. The application requires the minimal amount of setup in order to run.
3. The code is robust in recovering from errors and able to run continuously.
4. Proper programming practices are followed whenever it is possible.

1.1 Scope of the work

The work done in relation to one of the containers in this project is partially based on a group project[1], concerned with the acquisition of data from a web service in order to answer simple questions with SQL queries. For that project, the author of this thesis implemented the data gathering part using Python language with Selenium module. This individual work

[1]Group members: Jakub Sieńko, Kacper Garcon and the author, collaborating during an university course *Data Warehouses and Data Mining Systems*, conducted and supervised by dr Anna Gorawska, the promoter of this thesis

has been transfered and heavily modified in order to compose a container-based solution, with other containers written by the author from scratch for the purpose of this thesis.

In totality, the following parts were implemented:

- data gathering container,
- database manager container,
- data mining container,
- *nodejs* container with simple web application,
- logging service,
- flag management service,
- database management service,
- web scraping Selenium-based service,
- data management service.

Additionally, the author researched topics concerned with container orchestration as a simplification of system architecture, Selenium framework for dynamic web data retrieval, optimization of "big data" SQL queries and management of large datasets, web applications development in Node.js environment and other related fields.

1.2 Thesis contents

Chapter 2 is concerned with the changes in technology leading to various options of exploiting data availability on the Internet, while also deliniating the tools which are at the core of this project. It also explains the motivation for choosing such thesis topic.

Chapter 3 describes the domain of the problem with functional and non-functional requirements of the system. It also gives a high-level view of the data warehouse and the data inside it.

Chapter 4 contains technical information about the coordination of the containers, how they utilize the services and gives an overview of the data pipeline.

Chapters 5 and 6 are centered around the three main containers, taking a deep dive inside each one of them and following the data on its path from the website to market reports.

Chapter 7 reflects on the practical results generated from the data analysis, while constrasting them with initially posed questions. Additionally, the chapter describes the performance of the code under normal and abnormal conditions, and how the project changed with time.

Chapter 8 summarizes the effects achieved during the implementation of the system and formulates conclusions about possibilities of utilizing containerization to create and manage data warehouses.

Chapter 2

Project basis

"Data is the sword of the 21st century, those who wield it well, the Samurai."

— Jonatan Rosenerg, former Senior Vice President of Products, Google

In the contemporary world, the Internet facilitates the backbone of world-wide services, including e-commerce, transportation, entertainment and businesses. This relatively recent change presented an open world of possibilities, with new creative ideas emerging every year. In comparison to the reality of the past, the amount of information available at hand is unprecedently greater than ever before, constituting an advancement so significant, it can be argued to be as important as the invention of the printing press or even written language. Tapping into this immense ocean of knowledge is possible by utilizing adequate tools, like programming languages capable of producing specifically targeted scripts. This potentially is a fertile ground for automation; and to take advantage of it means to extend one's capability of processing information, possibly by orders of magnitude, to access previously unavailable knowledge.

2.1 Data on the internet

When the Internet (or the "network of networks") emerged in the United States in the 1970s it did cause a stir in the media, as it wasn't visible by the public until the 1990s. Today, more than 4.5 billion people are estimated to have access to the Internet. Not long after the Internet bubble burst in 2001[1] came "Web 2.0", a technological concept that put

[1] Speculation related to providing any services online, just to gain revenue from advertising on the Internet.

emphasis on social networks and user-generated content. Social media (Facebook, Twitter, and Instagram) gain on popularity and more websites are created every day. Since mobile phones can go online as well, the number of Internet users worldwide exploded from about one sixth of the world population in 2005 to more than half in 2020. At the same time, tools to extract data from the Internet are being constantly developed and outperformed by better new tools and approaches. This practically instant availability of any kind of information is practical for computers more than for humans, especially if they're guided to perform complex analysis and present the results in a simple way, bridging the gap between information chaos and clarity.

2.2 Data warehouses

Data warehouse is a data organization concept that originated in late 1980s in IBM [1]. Barry Devlin and Paul Murphy were trying to find a way to optimize the processing of data from common source to different destinations, called decision support systems. These systems were composed of software taking various data as input, and producing a metric for finding solutions to a given problem. One example is using a decision support system highlighting unusual areas of a brain scan from a MRI, for faster recognition of potentially malicious changes. Before the practice of data warehousing, multiple systems had to acquire data independently from a business source, process it and then perform needed analysis. However, this approach yeilds several problems, most obvious of which is computational redundancy and consequently wasting of resources.

Devlin and Murphy's idea was to find commonalities between different decision support systems, gather all the needed data at once, process it and then store it in a ready-to-use format. This way, any application could request a specific set of data, tailored for its needs, without the need to perform heavy computation every time. These datasets are called *data marts*, and the isolation of decision support systems from their individual data sources proved to be a robust solution, that has quickly been implemented in businesses around the world.

In my project I gather the data from various pages on the card market website, then I process and save it in a *.csv* file format. These files are then read and converted to dataframes, which are converted to tables in a database. In order for another process to use the data efficiently, it is transformed into various data marts inside the database, creating a data pipeline from the source to the final program, which performs analysis. This way the author is utilizing data warehousing concepts, by collecting the data from a single source (one process responsible for data gathering) and delivering it to two destinations (web application and data miner).

2.3 Docker containers

Containerization (or OS-level virtualization) is a way of isolating resources inside an operating system without using virtual machines (VMs), to create self-enclosed, lightweight executables. The key difference from VMs is that virtualization uses a hypervisor — software that hosts guest operating systems and distributes hardware resources among them. Any process running inside a virtual machine only sees the guest operating system. Meanwhile, containerization uses only the host operating system and a container engine (e.g. *Docker*). From the point of view of a process running inside a container, the directory structure may largely differ from what user sees on the disk. Container should have only the minimal number of libraries and dependencies required to run it. Generally speaking, containerization is a paradigm for the operating system kernel to allow many isolated *user spaces* to exist in a shared environment, which translates to container sharing the filesystem with the host operating system without conflicts. However, recreating the container may mean losing all of our data and starting fresh. To avoid that, methods for data persistence are available, two of which are most popular and used in this project:

- Docker volumes — internal Docker storage, which persists removing the container image if stated explicitly. These volumes are not seen by the host OS.
- Bind mounts — specified directories inside the container, that will correspond to actual directories on the host operating system. This technique is similar to mounting an USB device in a filesystem.

The goal of containerization is to deploy applications securely and fast, without worrying about OS compatibility. The act of abstracting our software from the host operating system allows containers to be portable and stand-alone. Additionally, one can orchestrate many containers to run together and share the OS resources in order to perform a common task. This creates a perfect opportunity to use containerization for an application managing data warehouse, since the data has to go through many different stages in a pipeline, which correspond to separate processes, each inside its own container.

2.4 Python language

Python is a high-level general-purpose programming language. It provides levels of abstraction from the machine architecture, so that the user doesn't have to worry about managing memory allocation or typing the variables. It can be used to develop applications in a myriad of domains. Its history begins in December 1989, when a Dutch programmer Guido von Rossum became working on a successor to the ABC language [4]. Released in 1991, with major consecutive versions in 2000 and 2008[2], it today became a robust language for just about anything, from statistics and modelling, to computer vision and creating web applications. Python is taught in schools as an entry-level language, while at the same time being used by NASA[3].

Python offers a variety of modules and packages (also called libraries), which are files of code written by other developers with collections of functionalities, for example to manage image files. Among the most popular libraries we have *numpy* — for managing matrices and multi-dimensional tables similar to MATLAB; *openCV* — for processing and analyzing images and videos; *requests* — a simple HTTP library for communicating with the server; and many more.

2.5 Justification of the thesis topic

In this thesis the author uses and draws from the technology described above, in order to build a coherent and standalone system, responsible for automatic data warehouse management. The theme of the warehouse is trading cards market and problems related to performing the right decisions as a buyer.

The cards market information is acquired using Selenium framework, all of the data processing is done in Python, with MySQL database to store the results and helper tables for further analysis. Data warehousing systems, with their multistage data processing, make it possible to isolate independent processes using containers. With each step running as a separate asynchonous application, the data flow is clear and easily handled; and failure of one container doesn't affect the others and its effect on the pipeline is minimized. Additionally, the proposed topic mirrors the author's interests in data processing and analysis, web scraping and containerization.

[2]Python 3, current version.

[3]`https://github.com/nasa/podaacpy` is used for crucial communications with Jet Propulsion Laboratory

2.5.1 Related works

:)

Chapter 3

Specification

The presented system requests pages from card market website and scrapes their content to build a knowledge database helping the user take better decisions as a buyer. The results are presented in a form of a simple web application, as presented on Fig. 3.1, as well as in the form of graphs and plots in the *analysis* directory.

Implementation of the thesis topic involves a containerization software — *Docker* — which hosts several subapplications written as semi-standalone scripts, each responsible for a part of the data pipeline. Using Python and its vast collection of libraries the author is handling the data scraping, intial cleaning and maintenance of the pre-stage database in compressed CSV files; as well as managing the MySQL database, creating helper tables, extracting new information and visualizing the data to answer user's questions. With JavaScript, SQL and HTML the web application is querying the database connected to a *Node.js* server.

3.1 Functional requirements

The system is required to do the following:

1. The system should gather cards, sellers and offers' data and keep it up to date
2. The system should allow the user to choose an expansion of cards to gather
3. The data gathering system should run continuously and gather data once a day
4. The gathering system should visit all card sites from specified expansion and save (a)

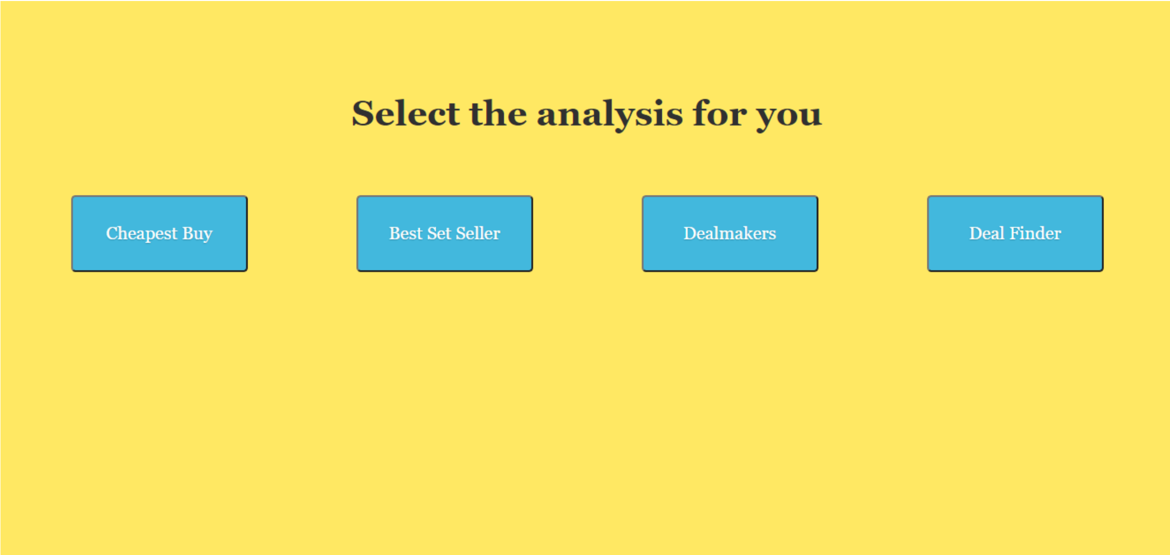

Figure 3.1: Simple web interface for performing analysis

card information, dynamic and static, and (b) full information about sale offers of this card to CSV files

5. The gathering system should visit profile pages of all newly-encountered sellers and save their public data to a CSV file

6. The gathering system should keep track of the date and save the date data into a CSV file

7. The database manager system should run continuously and update the database whenever new complete batch of data has been gathered

8. The data miner system should run continuously and create data marts every hour, given the database has the newest data

9. The web application should run continuously, recovering from errors and providing the user with at least four ways of getting useful information from the data

3.2 Non-functional requirements

1. The total time of data processing should be less than 6 hours

2. The system should be standalone (bootstrapping itself from *docker-compose.yml* and code), and cross-platform compatible

3. The gathering system should adapt the requests frequency to the server condition

4. No user input is required after running the system

3.3 Problem domain

The main goal of this project is to optimize user's card buying decisions basing on the data gathered in the warehouse.

3.3.1 Trading card games

Trading card games (*TCG*), also known as collectible card games (*CCG*), are types of card games combining the elements of strategic gameplay and features of trading cards. The first TCG was released in 1993 under the name Magic: The Gathering (*MTG*). The game, released by an eight-person company, was an overnight success, with over 10 million cards sold in just 6 weeks. Two years later, this basement business became a gaming corporation. Today, *MTG* is among the most popular TCGs with roughly 35 million players as of December 2018 [3].

During the game, the goal of each player is to reduce the opponent's life points by strategically playing cards from the hand, before the other one succeeds. However, each player can compile their own deck of 60[1] cards out of the thousands available. This makes every gameplay unique on a level which is fundamentally different from the classic cards. One doens't have to be an expert in the field to understand that the more cards one has, especially good cards, the higher the chances of winning. Thus, trading the cards becomes an aspect as crucial as the strategy itself.

One of the places where *MTG* cards can be bought is `www.cardsmarket.com/` — an online market with a myriad of cards from the game listed for sale, by users from around the world, majority of which lives in Europe. Users are divided into three categories: *Amateur*, *Professional* and *Powerseller*, depending on their setup and *modus operandi* (individuals, zealous hobbyists, card stores). To spend the least amount of money while collecting the most wanted cards, i.e. to optimize the shopping, one would have to analyze thousands of bits of data, from the average prices of all cards, to the velocity of sales and of restocking the virtual shelves.

[1]Some variations require a deck of 40 cards, which are selected from a random pool of cards

3.3.2 Market-related entities

Each card has a name, an expansion, which is a higher-level grouping of cards, and rarity. These elements do not vary with time. However, the number of cards for sale, and automatically calculated price statistics change every day, that's why these are stored in a separate table, with new rows every day. Similarly, the seller has a name, country of residence and optionally an address, year of registration and what type of seller they are (private vs professional); while the table with sale offers stores every card offer from every seller, from each day. To add a unified time dimension to the data, a date table is created with unique *date_id* designating the day, month and year of the datapoint collection.

When the data from the card market is collected, it is stored in five CSV files and one text file. The text file, named after set expansion, contains card names in the order of first visit and it's used to maintain consistent card-to-card progression between runs. Then, each CSV file represents one entity: *card*, *seller* (static entities), *card_stats*, *date* and *sale_offer* (dynamic entities).

3.3.3 Problem formulation

From the point of view of a card collectioner, every potential buy should be analyzed for better offers in order to minimize losses. The main goals of the user among buying needed cards at the lowest possible price, without sacrificing the various card qualities; finding the most price-slashing, discout-oriented sellers; discovering which cards are more discounted than others, and thus make a better buy opportunity; or minimizing the shipping costs by ordering a whole set of cards from a single seller, should one exist. Therefore, the main questions posed by the user should be answered accordingly:

1. Given a non-empty list of card ids (and card qualities: language, condition, whether it is foiled), find three sellers that sell that card at the lowest price.
2. Given a non-empty list of card ids (and card qualities) rank them based on price statistics, to highlight the most discounted cards.
3. Given a non-empty list of card ids (and card qualities) select users selling the maximum amount of wanted cards, ordered by the lowest total price.
4. Given wanted card qualities, find which sellers have a high amount of discounted cards and the biggest differences between the average market prices and their prices.

5. Additionally, present as much novel information and insights from the data as possible.

3.4 Use of Python

This project utilizes several crucial libraries, which are described in separate subsections below. The rest is briefly described in the following list:

- *checksumdir*: used for calculating a checksum of given directory, similar to a file checksum. It helps determine whether new data has arrived, is it complete and can it be passed further;
- *time*: used for measuring time between two points in code, sleeping (that is, stopping the execution) and getting information on the current date. Helps with scheduling the run until the day changes and measuring performance of various parts of code;
- *shutil*: used for creating and removing non-empty directories. Helps additionally isolate the shared data inside each container, so that it's invulnerable to changes in the original folder once running;
- *os*: used for managing local files and ensuring proper directory structure on the first run — creating shared folders for storing data, logs and flags.

3.4.1 Selenium and Beautiful Soup

These libraries make it possible to connect to any website and emulate user behaviour in an automatic way. Selenium is a framework responsible for creating a webdriver (virtual, in-code browser object) and visiting requested URLs, while looking for specific elements (like buttons for expanding the page) or deciding whether the response from the page is complete or it needs more time to load. Beautiful Soup on the other hand is a simple but powerful system for navigating HTML code. It provides support for finding particular web-page elements like *div* or *span*, by requesting their *class*, *id*, or any other attribute. Every HTML page is converted into a soup object, similar to a nested dictionary, which then can be searched using provided methods.

3.4.2 Numpy, Pandas and Scikit-learn

Numpy, briefly mentioned before, is a mathematical library which is a basis for many other libraries, such as Pandas or Scikit-learn. Pandas revolves around Dataframes and Series — two- and one-dimensional data structures similar to Excel or SQL tables. One of the advantages is the speed of processing; selecting tens of thousands rows out of millions, based on some condition, is almost instantaneous. Dataframes come with set of tools for their manipulation, i.e. aggregation functions, joins, concatenation, etc. When this data is passed, scikit-learn is responsible for creating statistical models trying to find correlations, predict outcomes, as well as provide decision support based on the data. It is arguably one of the most important libraries in machine learning domain.

3.4.3 Matplotlib and Seaborn

Matplotlib and Seaborn are two libraries for data visualization, where the latter is an extension of the former. Matplotlib gives basic plotting functionalities and proves to be a robust module, which can be used on its own. However, in order to reduce the amount of code written and standardize the outcomes, Seaborn is used to provide more complex visualizations without an extensive use of Matplotlib.

3.5 Data warehouse modelling

The data warehouse lifecycle beings when the data is collected. The pre-stage area encompasses what is written in the CSV files after a gathering run. These files correspond to entities from subsection 3.3.2 and to the tables presented below.

3.5.1 Base tables

After the data is gathered from the website, and stored in the CSV files, the database manager updates an image-based MySQL server running in Docker, creating the tables corresponding to each entity (seen in Fig. 3.2)— and data miner adds new tables as data marts for future use, and performs other analyses and visualizations.

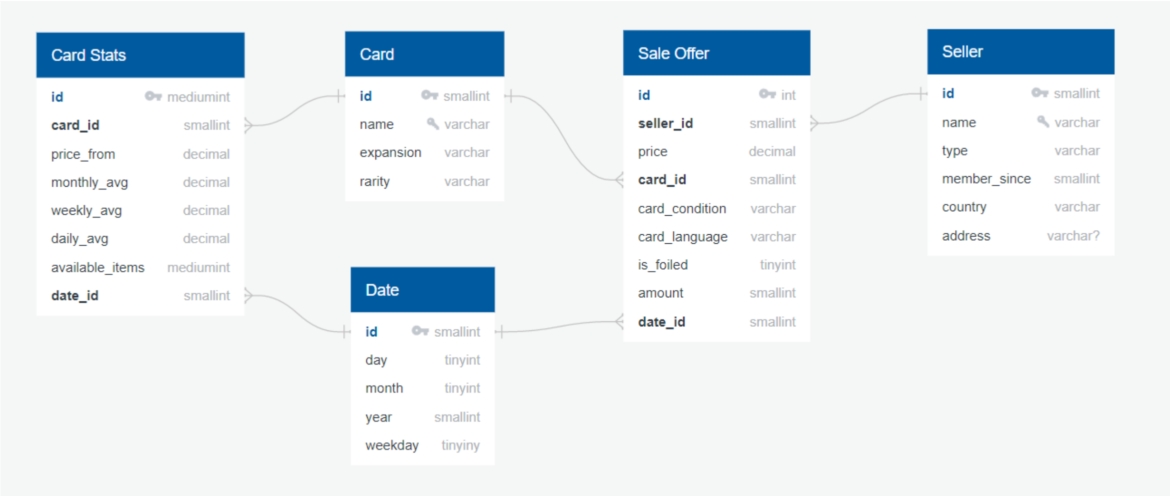

Figure 3.2: Database schema with the five base tables

3.5.2 Helper tables

The data miner creates new tables basing on the five elemental ones, in order to speed up the processing of SQL queries. The largest and most problematic table is the *sale_offer* with few millions of records, eight attributes each. To reduce the loading time, the newest day is selected as a separate table. Similarly, last two weeks are another temporal subset, extraction of which helps with performance.

3.6 Containers overview

The task of the *firefox_webdriver* container is to provide a constant and reliable access to a browser-like solution for an automated page navigation.

The *mysql_database* container hosts a database with persistent memory, storing the major part of the data warehouse and making it accessible to other containers.

The *data_gathering* subsystem automatically scrapes and collects the web data in the *data* directory, while *database_manager* takes this data and passes it to the database. It is important for the *database_manager* to be slightly delayed with respect to *data_gathering*, for reasons that are explained in section 4.1.1.

The *data_miner* and *node_server* containers are dependent only on the database, however here boot order and system state are not as important; these asynchonous containers perform their job properly when the data is ready and don't cause critial failure when it's not, which is enough for the application to run reliably.

The *node_server* container hosts a website served by a Node.js server, that presents the user with four functional analyses of the card market.

Each container has its own virtual storage, so to perform tasks on a shared set of data, both docker volumes and data mounts are introduced. Additionally, each container exposes one port internally to a port on the host, inside a network encompassing the whole solution.

Chapter 4

Project architecture

The described solution is composed of four directories with subprojects:

- data-gathering,
- database-manager,
- data-miner,
- server,

three auto-generating directories with program-related data:

- logs,
- flags,
- data,

and a docker-compose.yml file. Each of the four directories has its own Dockerfile, and acts as a build context for that container image. Additionally, two containers are pulled from the Docker image repository, the MySQL server (*mysql/mysql-server*) and Selenium standalone Firefox webdriver (*selenium/standalone-firefox*). Figure 4.1 shows the dependencies between containers.

4.1 Containers

Issuing the `docker-compose up` command builds each local application image from a respective Dockerfile, and aranges them according to the configuration in docker-compose.yml

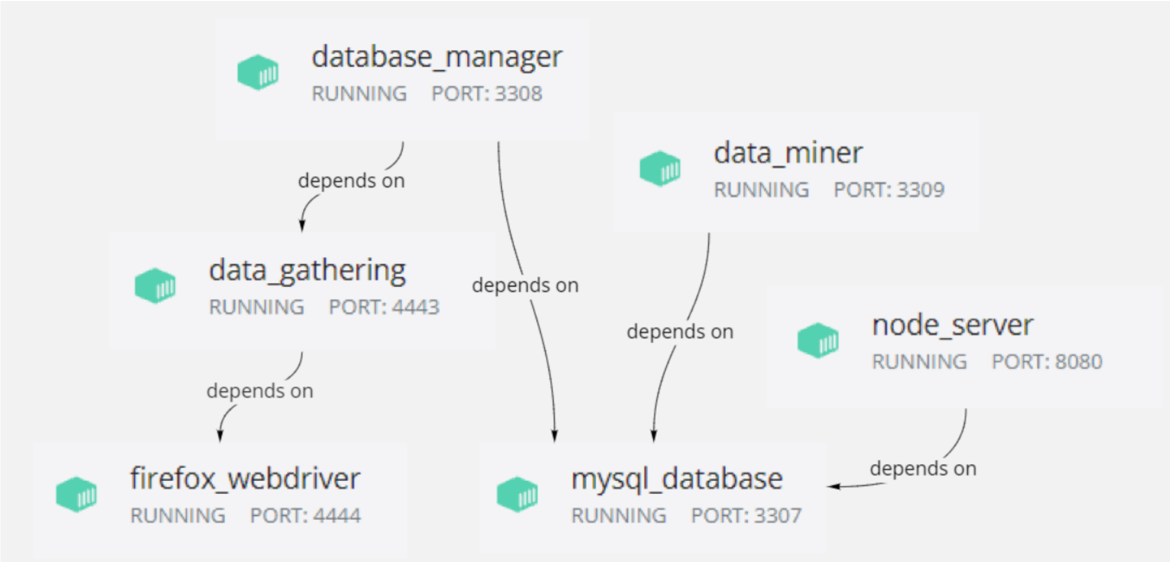

Figure 4.1: Boot order of containers (least dependant first)

file. All of the containers are booted to a shared network, with each container's name serving as an alias to be recognized immediately by other connected containers. Thus, every application exposes an internal port to the host via parameter:

```
ports:
  - "3308:3306"
```

This notation indicates that port 3308 inside the container maps to port 3306 on the host. Additionally, to persist data between system composition and decomposition, docker volumes and data binds are used:

```
volumes:
  - ./data:/data
  - ./logs:/logs
  - ./flags:/flags
  - pickle_data:/.pickles
```

Here, we declare that three directories relative to docker-compose.yml are supposed to be visible from the container's point of view as mounted in root directory. Any change to files inside the container will be immediately seen from the host level, providing the ability for several containers to work on a shared set of data. The pickles directory, on the other hand, is mapped to a *named Docker volume*, that is virtual space managed by the containerization software. These volumes are persistent and very fast, but there is no way to peek at their contents. Only the running container can acccess the data there, therefore it acts perfectly as a space for files that should only be accessed by given container.

All of the containers also restart automatically on failure and share a set timezone. To stop the operation of the system, one can stop the app in Docker Desktop, or stop any of the containers, or delete them altogether. However, images created during the build will still exist, so to fully remove the app one can issue a command `docker-compose down -rmi local -v`, which stops and removes all containers, removes all local images, networks and volumes.

- ***firefox_webdriver***
  The *firefox_webdriver* container exposes port 4444 to send the traffic to the server through. It has no persistent memory and in general behaves like an abstract version of a Firefox browser running in the background, operated by any script that connects to it.

- ***mysql_database***
  Container *mysql_database* exposes hosts a database with persistent memory, storing the major part of the data warehouse and making it accessible to other containers.

- ***data_gathering***
  This subsystem automatically collects the web-scraped data in the *data* directory

- ***database_manager***
  This takes this data and passes it to the database. It is important for the *database_manager* to be slightly delayed with respect to *data_gathering*

- ***data_miner***
  These containers are dependent only on the database, however here boot order or system state are not as important; these asynchonous containers perform their job properly when the data is ready and don't cause critial failure when it's not, which is enough for the system to run reliably.

- ***node_server***
  These containers are dependent only on the database, however here boot order or system state are not as important; these asynchonous containers perform their job properly when the data is ready and don't cause critial failure when it's not, which is enough for the system to run reliably.

4.1.1 Container orchestration

The containers are all started when the `docker-compose up` command is issued in a root directory of the project. This causes the containers from external images to be loaded first, based on the dependency tree. From there, more control over the mutual cooridation is available via healthchecks or timing. Out of simplicity, the second approach has been selected for this thesis; i.e. the gathering project is initialized with an immediate 10 seconds delay, so that the webdriver is surely ready before a connection attempt takes place. Similarly, the database managing part is started after 20 seconds, so that the previous container can assess whether there is some new data and any action is needed, or that data can be verified, and marked as ready to be updated to the database. The data mining part is delayed 30 seconds, in case the database is in pre-update state, and other services will start to act on it. However, the most important element of orchestration is shared access to the flags directory, which stores two files with directory checksums, calculated from the ready and complete data after a gathering run. This checksum becomes the new standard and an indicator that the database contents may be outdated.

4.2 Services

The three programs written in Python share similar functions. These are grouped into **services** by their main utility and by needed external libraries. Each directory contains a *services* module with the following:

- *logs_service* — responsible for creating a container-specific logs directory with a daily log file and optionally run log file, general logging to file and console. Libraries needed: **os**, **time**.
- *flags_service* — this service creates the flags directory, sets up checksum files and provides an interface for everything related to detecting differences in directories or checking the current stage of the data. Libraries needed: **os**, **checksumdir**.
- *data_service* — handles everything related to CSV files, translating the scraped data into attributes, loading and saving dataframes, pickling and unpickling the data, all data manipulation pre-mining and ensuring the completeness and correctness of the data. Libraries needed: **os**, **sys**, **shutil**, **time**, **pandas**.
- *web_service* — interface for connecting to the webdriver and navigating the pages,

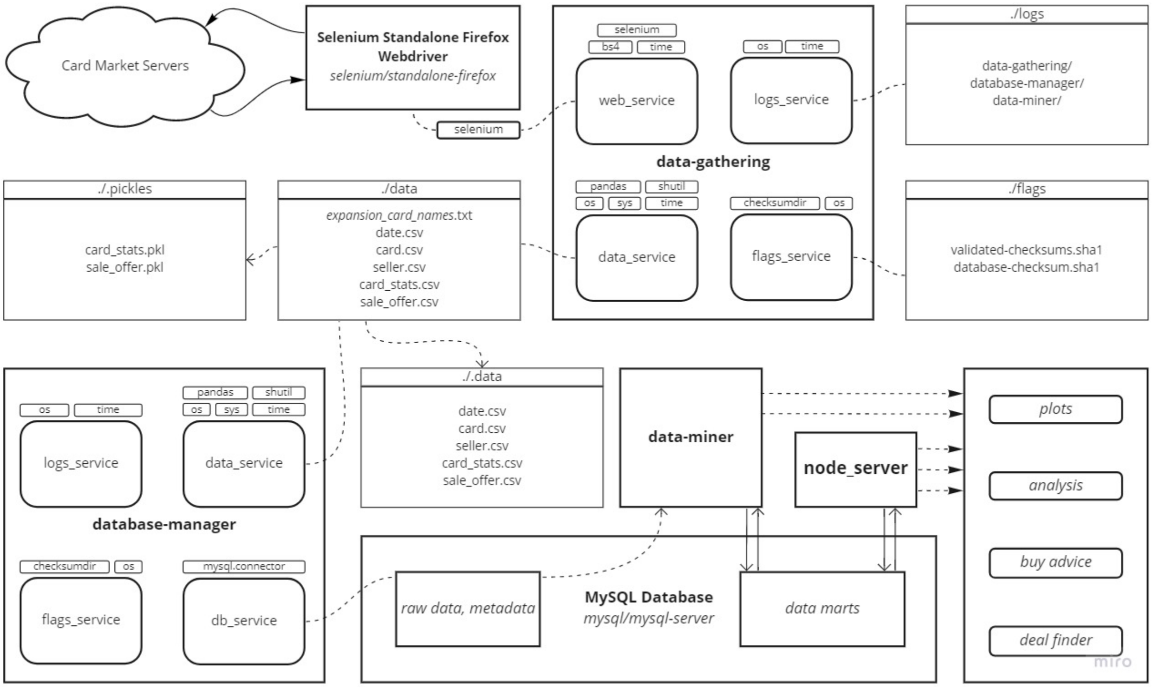

Figure 4.2: Data pipeline from the server (left-top) to results (right-bottom).

finding appropriate elements on the website, implicit and explicit waiting for response, page validation and so on. Libraries needed: **time**, **bs4**, **selenium**.

- *database_service* — manages connection to the database, setting up the tables, updating the schema with new data, as well as creating helper views and additional tables in the data mining part. Libraries needed: **mysql.connector**.

4.3 Data pipeline

As presented on Fig. 4.2, the data comes from the web market servers and, through Selenium, is gathered and processed into five CSV files, with intermediate files in pickle format, inside the container. Then, the directory is cloned and the entities from files are uploaded to the database. New data is generated from the base five tables and exploratory data analysis is performed. The end results include the *analysis* directory with graphs and plots, and the web application providing four analyses on the data. The speed of the consecutive steps were measured from the logs and were found to be as follows:

- Scraping the website data: around 1–2 hours when all sellers are saved, otherwise more than 6 hours

- Validating the data: around 1 minute
- Updating the database: around 5 minutes
- Creating helper tables: around 1 minute
- Performing the analysis: around 2.5 minutes

4.4 Technology stack

The application was created on Windows 10 Enterprise, with Docker Desktop version 4.4.4 installed. Data gathering and database managing containers are based on Python image *python:3.8-slim*, while data miner uses *python:3*. The server image is *node:16-slim*, Firefox webdriver is given by *selenium/standalone-firefox* and the MySQL database comes from the Docker image *mysql/mysql-server*. Both the project and the thesis were developed in Visual Studio Code 1.64, the latter with *latexmk* (*lualatex*) recipe. Git and GitHub were used for version control.

Chapter 5

Data gathering implementation

As mentioned before in this thesis, the *data_gathering* container consists of a Python project, which main function is to collect card market data. Static data is acquired once, dynamic data daily, and the validity of the data is ensured after every run. The user can alter the program's operation by changing values in the configuration file, and the responses are provided in console and log files. The core part of the implementation in contained in *web* and *data* services. After this container completes its objective, the CSV files in data directory are updated with the newest information available, and their checksum is calculated to be retained as a marker for the verified dataset.

5.1 Program configuration

The configurable global variables are stored in file *config.py*. There are three subgroups inside:

1. Variables for user custom program configuration.

   ```
   START_FROM = 1
   FORCE_UPDATE = False
   EXPANSION_NAME = 'Battlebond'
   ```

   The user may want to renew the daily data when suspecting incompleteness or start the operation from a specific card id (for example, starting from 128 when the previous run crashed after card 127). Moreover, the user is provided with the ability to change the expansion, so that efectively all cards from the game could be gathered.

2. Variables connected to a single run of code.

```
DATE_ID = 0
MAIN_LOGNAME = ‘other_main.log’
RUN_LOGNAME = ‘other_run.log’
```

These values are determined at the start of the run (whenever a new date has been detected). Log filenames are unique to the run and are used by the logging service. Shared date id is crucial for all operations of the program.

3. Fixed variables.

```
CONTAINER_DELAY = 10
NAME = ‘data-gathering’
BASE_URL = ‘https://www.cardmarket.com/en/Magic/Products/Singles/’
USERS_URL = ‘https://www.cardmarket.com/en/Magic/Users/’
WEBDRIVER_HOSTNAME = ‘firefox_webdriver’
HEADERS = {"date": {"id": "int", "day": "int", ...}, "card": ...}
```

Fixed variables are used for internal program logic. They are aggregated in a single file to be changed with convenience and to be available wherever needed by importing the file. They set up the website’s paths, connection details and information about entities columns and data types.

5.2 Used services

The program uses four services:

- *Web service.* Responsible for handling the connection to the remote webdriver in another container. It serves as an interface for acquiring the cards’ names from specified expansion; loading card and seller pages; clicking the *Load more* button to expand the page fully; getting website’s source code in form of a Beautiful Soup object; cooling the connection down to prevent HTML response status 429[1]; and verifying the completeness of loaded pages.
- *Data service.* This biggest singular service (over 700 lines of code) utilizes the *Pandas* library to take apart soup objects and transform the information they hold into data records in csv files. This module also provides quasi-database-like functionalities,

[1]Too Many Requests — The user has sent too many requests in a given amount of time ("rate limiting").

like finding card's id by its name or checking whether a particular sellers or card statistics are already saved. Here one can also find the use of *pickle* data format, which is astonishingly fast in comparison with other data extensions, as well as the logic responsible for pickling, unpickling, updating and verifying the data.

- *Flags service.* The main use for this service in *data_gathering* container is to signal whenever a dataset has been checked (collected data is tested against various inconsistencies, for example the number of card statistics from each day should equal the number of cards, etc). When the data appears to be complete, the checksum of the entire data directory is calculated and saved in a shared file, for the next stages of the data pipeline to access and confront the datasets against.

- *Logs service.* This simple service is a wrapper for the print command. It uses the current time and values from the configuration file to display the state of the program in the console and generated log file. Each container generates a log subdirectory for its own purpose.

To keep the operation of the program simple and fast the author decided to prefer modular programming over object-oriented programming. OOP approach would probably yield some simplicity with methods related to the entities (e.g. *Card* class, with scraping the data, importing, exporting and updating inside the class body as methods), but it would entail sharing the class among containers, which would require all of the containers to have the same external libraries installed, which would reduce their build time and performance. Using functions on module-levels, with separation of resposibility between modules ensures minimal redundancy in importing external libraries, as well as provides fast and readable code.

5.3 Local directories

The container is given access to three host directories via bind mounts: **data**, where the csv files are stores; **logs**, with data-gathering subdirectory filled with daily and run-level logs; and **flags**, containing SHA-1 formatted file with a list of checksums of verified datasets.

There is also a docker volume named *pickle_data*, which maps to a folder named **.pickles** in the container root directory. This space is a storage for the data in pickle format, used only by that container during a gathering run.

5.4 Run scheduling

The program begins by booting the directories up if they don't exist and determining the current date. If it's not added to *date.csv*, new entry in the file is generated and its date id is returned. Otherwise, just the date id is retrieved from an existing date. Then, the execution is halted inside a scheduling loop, until the conditions for running the gathering procedure are met.

```
# Setup
logs.setup_logs()
data.setup_data()
flags.setup_flags()
data.add_date()


# Time the program execution
data.schedule_the_run()
```

As seen on Fig. 5.1, the run scheduling function checks whether the configuration flag `config.FORCE_UPDATE` is set or whether it's a first gathering run in the environment, which results in immediate return of control to the main function.

If that's not the case, the data directory checksum gets calculated and compared to the validated hashes. In case of occurence in the control file, the program sleeps for 60 minutes, as current data has been validated to be complete. When there is no entry in the *validated_checksums.sha1* file, the program checks whether today's data is complete. If not, the operation proceedes to gather the data, otherwise it saves the checksum as verified and waits 60 minutes. Every time after sleeping, the date is checked and date id is chosen accordingly.

5.5 Data pickling and validation

Initially the data was stored in raw csv files, with each new entry being appended to the end as text. This approach proved to be conceptually simple, but not very scalable, as it yielded higher and higher wait times for opening and closing files with hundreds of thousands of rows. Currently, the *sale_offer.csv* file stores roughly 10 million rows of data, but the total waiting time for opening, reading, writing and closing is greatly reduced. This is done by using the pickle data format with very fast r/w speed. Between the runs,

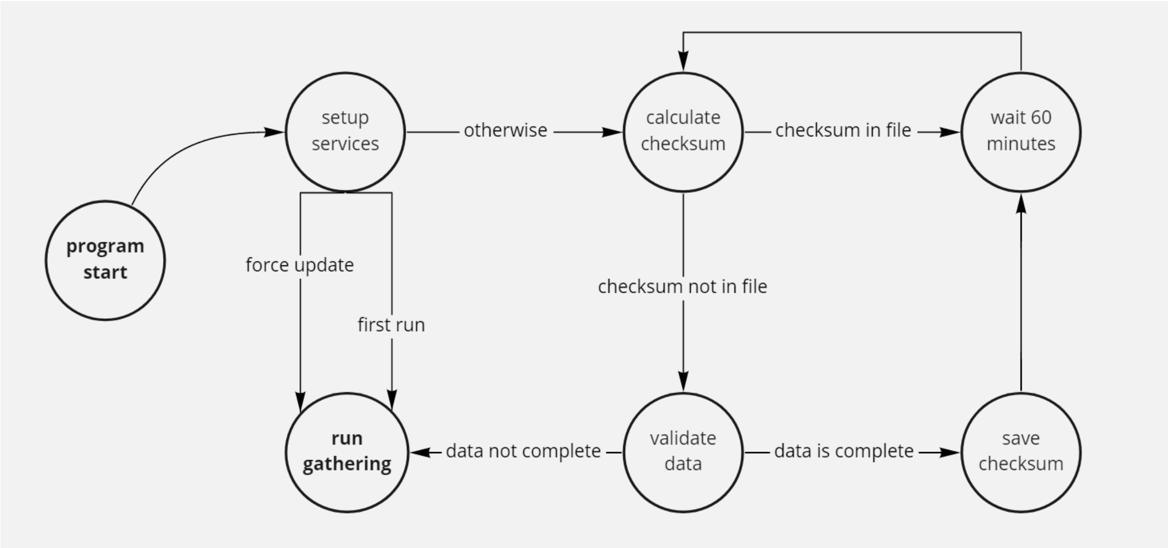

Figure 5.1: Program states in data gathering scheduling

data is kept in gzip-compressed csv format, to ensure minimal disk usage. However, as the run begins, some of the files are read into dataframes and converted to intermediate *.pkl* files. Then, whenever a pickled entity is needed, the program knows to read the pickle file instead, which loads about 100 times faster. When the program is finished, the pickled entities are transformed back to csv format. This step is the biggest hazard of potential data loss, as hours of computation kept in few, hidden files replace the previous data in a matter of minutes.

The pickle data format posed another challenge: unexpected errors concerning pickling directories and iteratively growing a pickle object. Unfortunately, the approach to acquiring sellers and cards was exactly that; a dictionary with pairs of attirbutes and values represents a single seller and to keep all entries, a list is used. Unpickling the whole object to add one seller and then pickle it back was causing the data to become unreadable after few hundret iterations. The speed of operating on gzip-compressed CSV files with less than 10 thousand rows was acceptable, especially taking into account that sellers are only added when they are not in the table, but sale offers and card stats are updated every day.

5.6 Loop over cards

The program works by querying the local TXT file with names of the cards and creating proper card urls according to a template. The webdriver loads the page with explicit wait time equal to 1.5 seconds, which gives this much time for the request elements to load. The

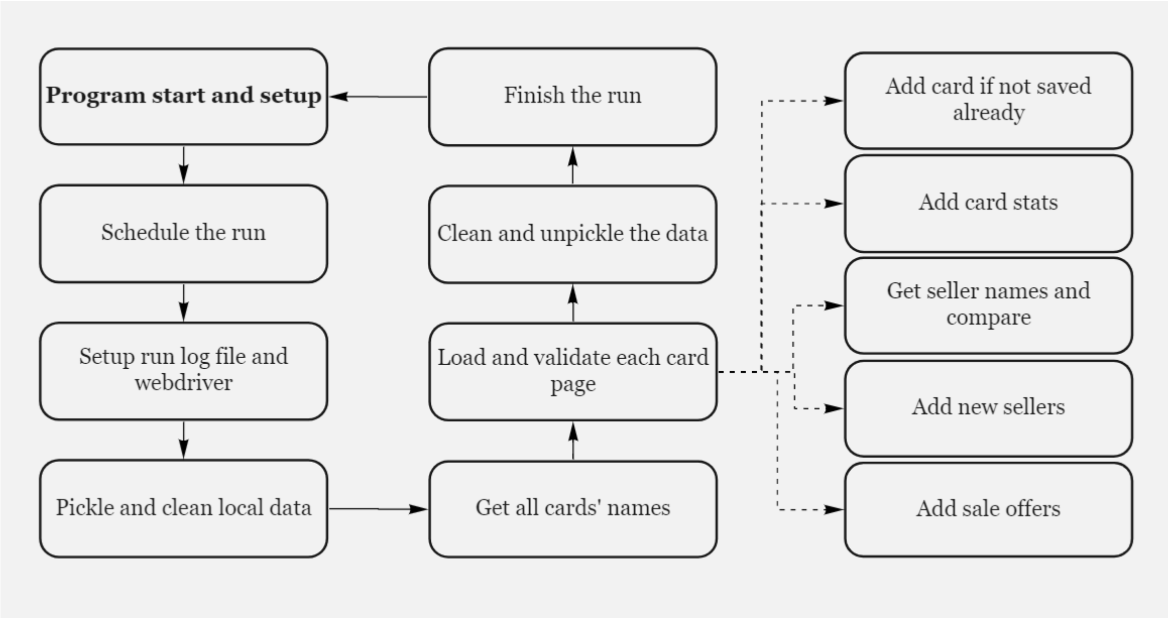

Figure 5.2: Program states in data gathering scheduling

card page features all the card information (Fig. 5.3) and a list of sale offers which can be expanded by clicking a "Load more" button on the bottom of the page (Fig. 5.4). This button is one of the reasons for the use of Selenium framework as opposed to *requests*. We need dynamic control of the webpage to interact with the button, until all the offers are loaded. After each click, the program waits up to 1.5 seconds.

Getting this page is the most network-heavy operation of the code, thus we save the complete webpage as a soup object to be deconstructed later. Each card soup is passed to appropriate methods to extract the card statis information and statistics, as well as the list of offers. After the card processing is completed, a START_FROM counter is incremented, so that the program can pick up from the last completed task (provided it doesn't fail into restart). After all card pages have been visited, acquired and processed, we clean and unpickle the data, then the program control comes back to the scheduler, where the data can be checked for completness and validated.

5.7 Soup decomposition

The card page source, packed in a soup format, is used to get the name of the card. If no such card exists in our records, it is added, together with the expansion name, rarity and a unique id. The time-related card characteristics are acquired every time. They need to be

Rarity	
Number	213
Printed in	Battlebond
Reprints	
	Show Offers / Show Versions
Available items	1042
From	0,40 €
Price Trend	1,43 €
30-days average price	1,32 €
7-days average price	1,26 €
1-day average price	1,68 €

Figure 5.3: A *div* element holding card info from its webpage

Seller		Product Information		Offer	
70	hobbitfred	NM		0,40 €	1
447	stamosm	NM		0,50 €	2
3	JBL88	NM		0,50 €	1
65	happysquid	GD	Some small marks on front	0,59 €	1
8K	ggmagic	GD		0,60 €	1
110	maxsilver	NM	#3	0,60 €	1
99	elliot1990pl	EX	Selling out collection	0,60 €	2
278	McFreud	NM		0,65 €	1
105	bendu71	NM		0,65 €	2
40	JJ-Finkleton	NM		0,65 €	1
4K	davomon	NM		0,69 €	2
511	Tzigouli	NM		0,70 €	1
140	Sadquatch	NM		0,70 €	1
98	ClemFr	NM		0,70 €	2
1K	Haaggen	PO	Bend on Middle and Bottom - Totall...	0,75 €	1
955	Flo33333	NM		0,75 €	2

Figure 5.4: A table of sale offers on given card page

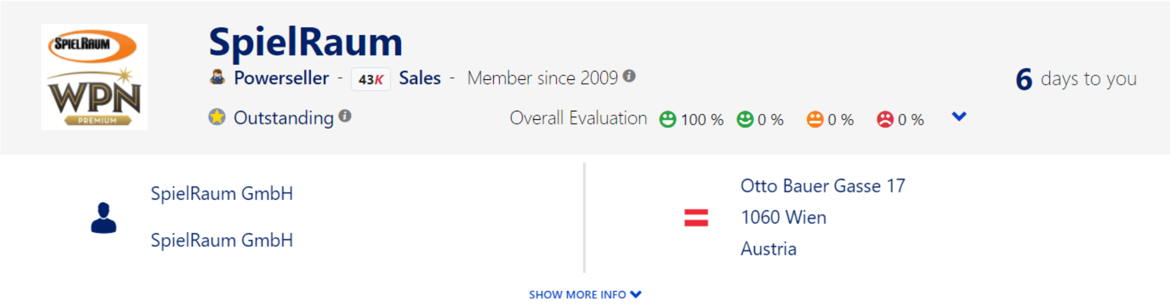

Figure 5.5: A *div* element with seller information on their webpage

updated daily and the most recent data has priority. In case of second run of the day the saved card stats are dropped.

5.7.1 Sale offers table

A list of sellers names if acquired from the source code of the page, along with the list of prices, card conditions and other sale-related information. The seller names are stored in a set, from which we substract all the names of already known sellers. The remaining names are to be saved, so appropriate user page URLs are generated according to a template. Then, every new seller page is visited and data about them is saved to the seller collection (Fig. 5.5). Due to sellers being added cumulatively into the dataframe (use of `append(seller: dict)`) caused pickles-related fiasco.

Updating sale offers is possible because the separate lists of seller names, prices, conditions and other, can be zipped into a dictionary. This way, we're scanning the table vertically and concatenating the attributes together to form a new set of records to be added to the sale offers file. If the offers are already saved that day, the old ones are dropped in favor of the new ones. Complete update however, happend only at the end of the run, when the pickled data is used to replace the pre-run data.

5.8 Data pipeline output

When the program is finished we have a complete list of card names, separate CSV file of static cards information with their IDs. The file with card statistics will have 264 new entries corresponding to each card and any new sellers will be saved in their CSV file. The sale offers file is going to have about 90000 more entries, representing the offers placed by sellers for each card. All the data is stored in five compressed CSV files and one TXT file. The data is ready to be checked for completion (e.g. to detect 263 cards gathered instead of 264). When the data is verified to be complete, a checksum of the data directory is calculated and saved in the validated-checksums.md5 file, so that other containers can recognize this dataset as ready to be pushed forward the data pipeline.

Chapter 6

Data warehouse management and data mining

Up to this point, our data was stored in the *data* directory in CSV format. The role of the database managing container is to take this data, ensure its correctness, and upload it to a MySQL database. MySQL was chosen because of relative simplicity paired with decent scalability. Additionaly, the only library used in relation to the database is *mysql-connector-python*.

A database is introduced because although having the data stored securly in compressed files on the host filesystem may seem good, we would like to be able to use the data to gain information in the problem domain. Having created the tables in our database, we can use SQL to retrieve information from them; and powerful, multi-stage queries may provide unexpectedly novel knowledge from simple entities.

A data warehouse is a subject-oriented, integrated, time-variant, and nonvolatile collection of data in support of management's decision-making process [2]. In order to be congruent to the definition above, presented system must generate a collection of data that is:

1. subject-oriented — related to questions in a single problem domain, e.g. gathering vending statistics and product information in supermarkets to improve sales through better advertising.

2. integrated — combined from possibly several sources into a single, uniform warehouse, which can be accessed by any department within the organization, and the data is ready to be structured into spreadsheets or tables.

3. time-variant — contained and consistent within a time period; when loaded daily, weekly, or hourly does not change within that period.

4. nonvolatile — locked from changes to the gathered data, which once entered into the warehouse, does not change.

6.1 Database managing

The role of managing the connected database with regards to updating the data falls to the *database_manager* container. It's internal structure and configuration is similar to the *data_gathering* container, described in chapter 5.

6.1.1 Program configuration

The configurable global variables are stored in file *config.py*. There are three subgroups inside:

1. Variables connected to a single run of the code.

   Database connector and MySQL connection configuration is stored here, together with names of log files and the checksum of data directory.

2. Fixed variables.

   ```
   CONTAINER_DELAY = 30
   NAME = 'database-manager'
   ```

3. SQL static queries.

   This section has two variables: DROP and CREATE which are dictionaries holding pairs of table names and appropriate SQL queries to drop and create them.

6.1.2 Used services

The program, similar to data-gathering program, uses four services, three of which are shared among both of them with minor changes.

- *Database service.* Responsible for handling the connection to the database. It provides access to the data, as well as functions responsible for updating the tables.

- *Data service.* Here the service is 53 lines long, with functions detecting the presence of flags file, isolating the data into a Docker volume, loading SQL entities into a Dataframe, and decompressing the CSV files into a container directory from which MySQL can load all records simultaneously.

- *Flags service.* This service provides the utility for managing datasets versions across containers; here we use it to check whether current dataset is indeed complete (in *validated-checksums.sha1*), then to see whether it differs from the one that is currently in the database (*database-checksum.sha1*). If it detects a disparity, the new data will be loaded, and its checksum saved to the latter file.

- *Logs service.* The use of this service is the same as described in the analogous section 5.2 in the previous chapter.

6.1.3 Run scheduling

After the initial setup, the program enters a control loop deciding whtether to commence an update or not. First, it compares the checksums of local files and database content. If the newest data is already in the database, the program sleeps for 30 minutes. Otherwise, it tries to verify the new data by checking it against the *validated-checksums.sha1* file. When the dataset is concluded to be new, but not yet verified, the program waits 15 minutes. After uploading the data, a 60-minute pause is initiated and the program begins the check again.

6.1.4 Tables from the staging area

When the data is gathered by the first container, it holds information about five entities: *date, card, seller, card_stats* and *sale_offer*. The data types of their attributes are dictated by the type of information they convey and are defined as described in subsection 3.5.1.

In order to upload the data to the database, uncompressed CSV files are copied to */var/lib/mysql-files* directory inside the container. This allows for the `LOAD FILE` query to be executed, drastically improving the processing speed.

6.2 Data mining

When the data is in the database, it can be used to gain insights about issues that concern us. Large tables can be reduced if the historical data is not needed every time, e.g. the *sale_offers* table, having 90 thousand new records every day and over 10 million overall, can be used to extract just the last two weeks of sales. This two-week time period information is needed by other queries and creating such a sliced table improves the access time greatly. The other increase in speed comes from a similar table, where only the sale offers from the current day are stored.

The data is also read into the mining module and transformed into Dataframes, table-like objects provided by the *pandas* library. These provide a multitude of transformations and method to be applied to the data, which the author uses to extract valuable information about the problem domain.

6.2.1 Program configuration

The config.py file holds variables related to the database connection and log files, similar to the *database_manager*. Fixed variables comprise container delay, here set to 60 seconds, and the project name. There is one control variable to be set when the *sale_offer* table has more than 10 million records, to change the data retrieval method.

6.2.2 Used services

- *Database service.* Responsible for handling the connection to the database, checking its state, the newest date and creating the helper tables through SQL queries.

- *Mining service.* Loads the data saved in the database and performs exploratory analysis, saving the results to the *analysis* directory.

- *Flags service.* Here the service is used to ensure that there is no new verified dataset to be uploaded to the database — in that case the program waits 5 minutes for the database manager to load the data.

- *Logs service.* The use of this service is the same as described in the analogous section 5.2 in the previous chapter.

6.2.3 Run scheduling

The run is halted for 5 minutes, whenever new data is about to be uploaded to the database. When the *database_manager* container's job is done, the database connection is established and the number of tables is checked. If it's less than 5, the analysis cannot be performed, and the program is set up to wait 120 seconds and query the database again. Should the number of tables be correct, the data transformation run is commenced. After finishing the program waits 60 minutes and begins the routine again. This way, data mining is not dependent on the database content, which may change 23 hours within a day, resulting in 23 correct separate analyses.

6.2.4 Data analysis

Apart from creating the helper tables, an analysis is performed on the collected data. Appropriate directory is created and the five dataframes undergo many transformations to extract the following information:

- actual rarity, number of cards in each rarity class;
- amount of sellers with public address, by seller type;
- site popularity from the number of new users each year;
- number of new sellers by year and seller type;
- ranking of countries with the most sellers;
- price statistics trends, average price history;
- average price statistics distributions;
- the number of available marketable items over time;
- new attribute indicating market rarity, inverse ratio of card availability to mean availability;
- heatmap of correlations between prices, available items and the calculated market rarity;
- a plot of each card's 50 best sale prices and when they occured with the best 30 and 10 offers marked differently (in total 264 plots in the *analysis/cards* subdirectory).

Chapter 7

Results

The data gathered, cleaned and stored in the previous parts of this project is now transformed, in order to be a foundation for discovering new knowledge through the web application interface. This simple subproject was written without much giving much attention to the front-end development or any user experience whatsoever. Its purpose is to provide a user-friendly communication with the database, narrowed to the specialized queries for answering the users questions:

1. Cheapest Buy: targeting three sellers which sell the provided card or cards for the lowest price (Fig. 7.1).
2. Best Set Seller: finding a seller with the most provided cards on sale, ordered by the lowest sum (Fig. 7.2).
3. Dealmakers: identifying those sellers, whose offers stand out as the most discounted among others (Fig. 7.3).
4. Deal Finder: ordering the provided cards from the highest discount available to the card with most ordrinary price (Fig. 7.4).

7.1 Data analysis

The end results of the data analysis are plots in the *analysis* directory. Figures 7.5 and 7.6 show the number of sale offers and price trends versus time, respectively. Another example is the site's popularity presented via the number of new sellers per year (Fig. 7.7).

Figure 7.1: Selecting sellers that sell the given card for the lowest price

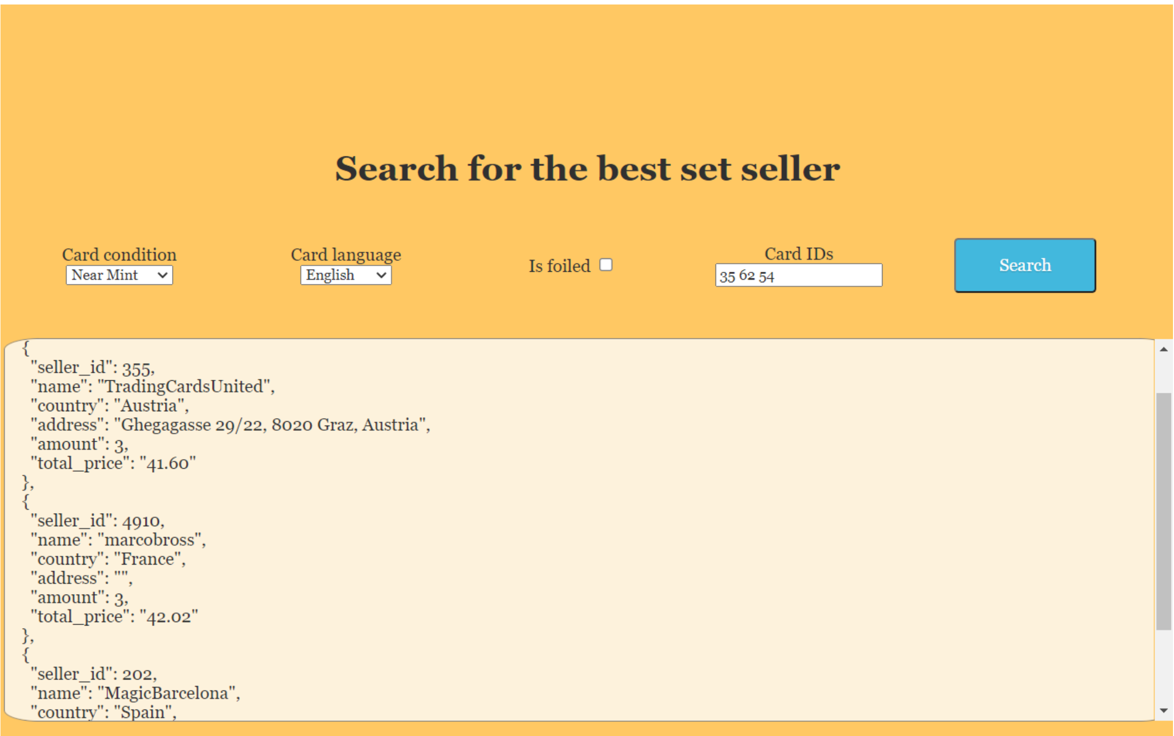

Figure 7.2: Finding sellers selling all three cards for the lowest total price

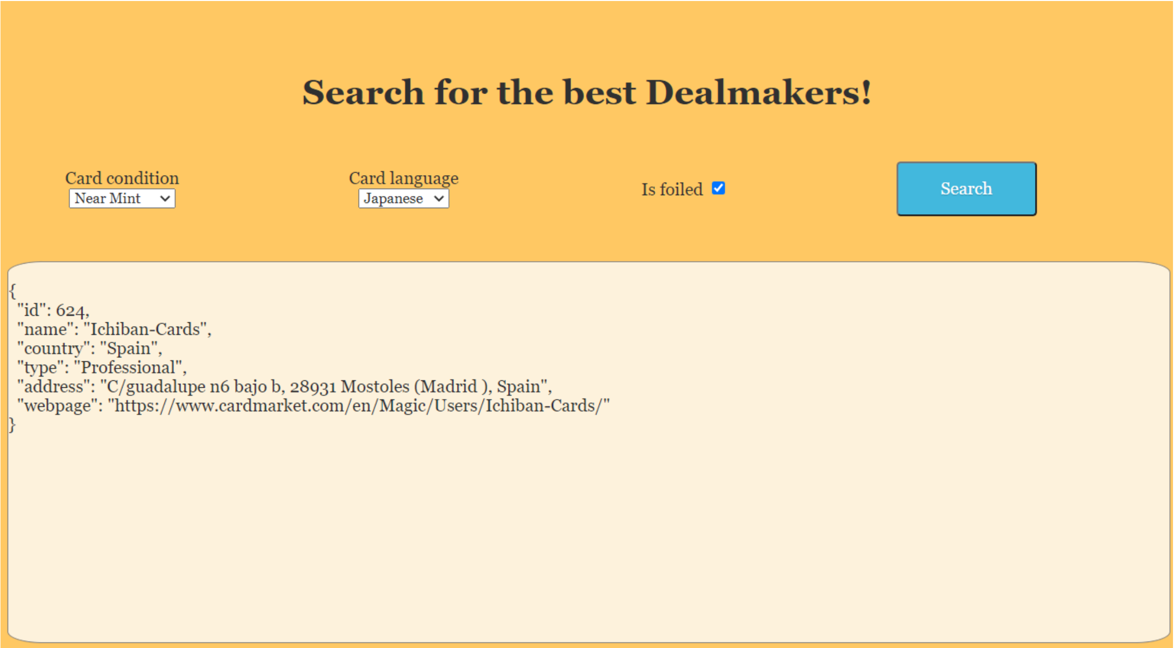

Figure 7.3: Finding sellers with the highest discount score

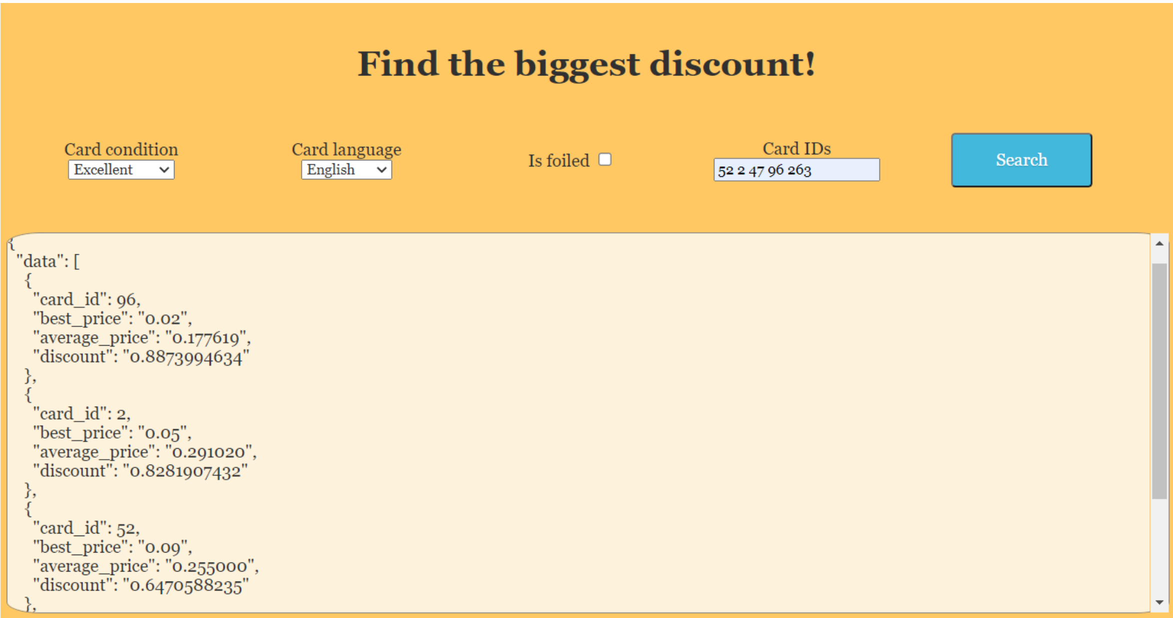

Figure 7.4: Ordering cards by the discount available

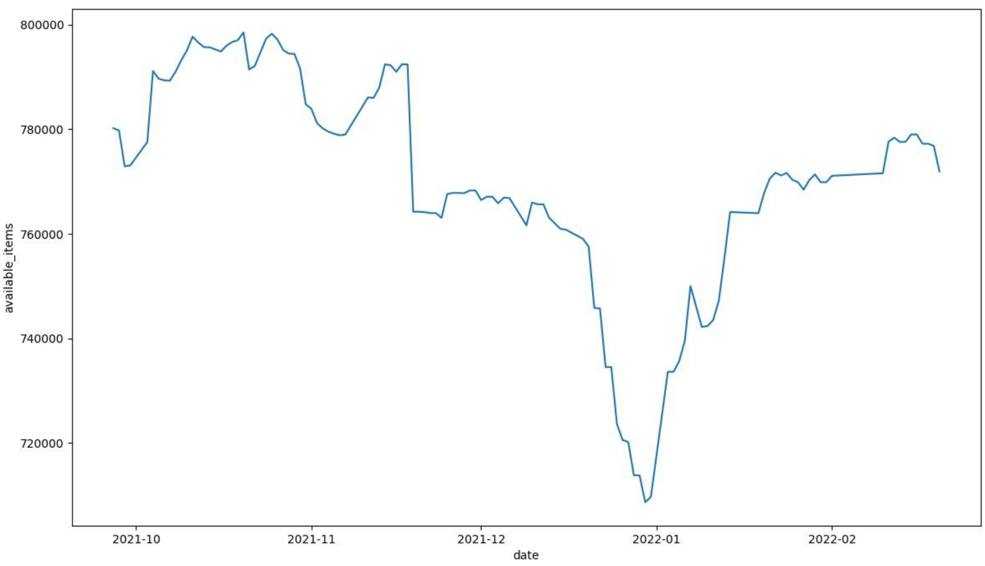

Figure 7.5: Number of items available for sale over time

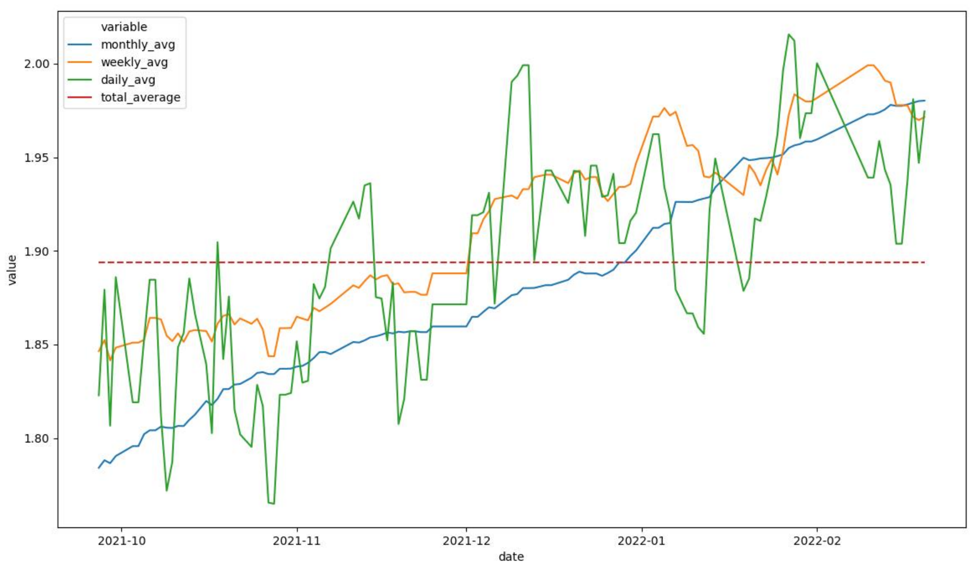

Figure 7.6: Price trend between October 2021 and February 2022

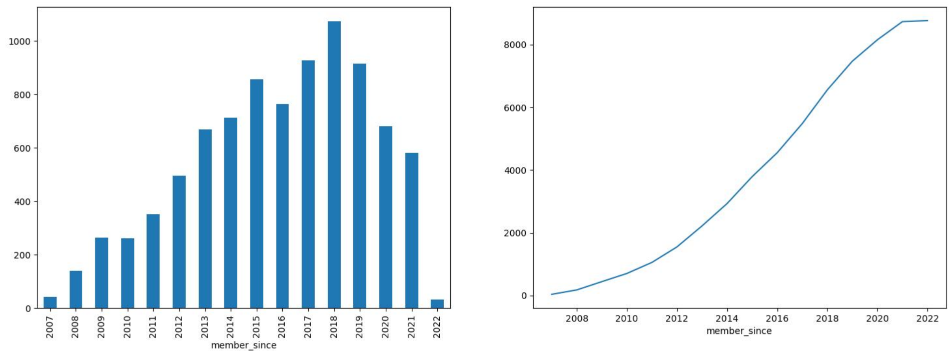

Figure 7.7: Site popularity based on the number of new users per year

As part of the analysis, scatterplots of best cards' prices are generated for each card individually. This help the user understand the price trends of the card by analyzing when did the top ten, thirty and fifty best deals occured, and what was the price. This way, without using any machine learning algorithms, we can provide a simple functionality for predicting the cards next *good deal* which happens inside the user's brain. For example, looking at Fig. 7.8 and 7.9 we conclude that one of the cards will probably be soon available at around €2.30, while the other is clearly rising in value and a price under €11.00 is not likely to occur anytime soon.

7.2 Running on Fedora

The system was successfully ran on Fedora 35 operating system, using virtualization software (*Oracle VM VirtualBox 6.1.26*). After updating the system using *dnf* package manager and installing *docker* engine (version 20.10.12) and *docker-compose*, the code was copied from private GitHub repository and executed. In Fig. 7.10 we can see that *docker* is installed and running, which entails issuing the `docker-compose up` command will build and run the application. The proper behaviour of the system is manifested in the scenario where a complete and valid dataset has a changed checksum, because of the migration to another operating system. We can observe the containers starting in Fig. 7.11 and the discrepancy between database checksum and files checksum is detected. The database manager will halt for 15 minutes, as it waits for the data gathering module to finish its job and validate the files. Since only the checksum changed, the data is recognized to be complete and is then validated and the new checksum saved to file. As presented in Fig. 7.12, during the next run we get an information that the dataset is already validated; now the dataset is validated,

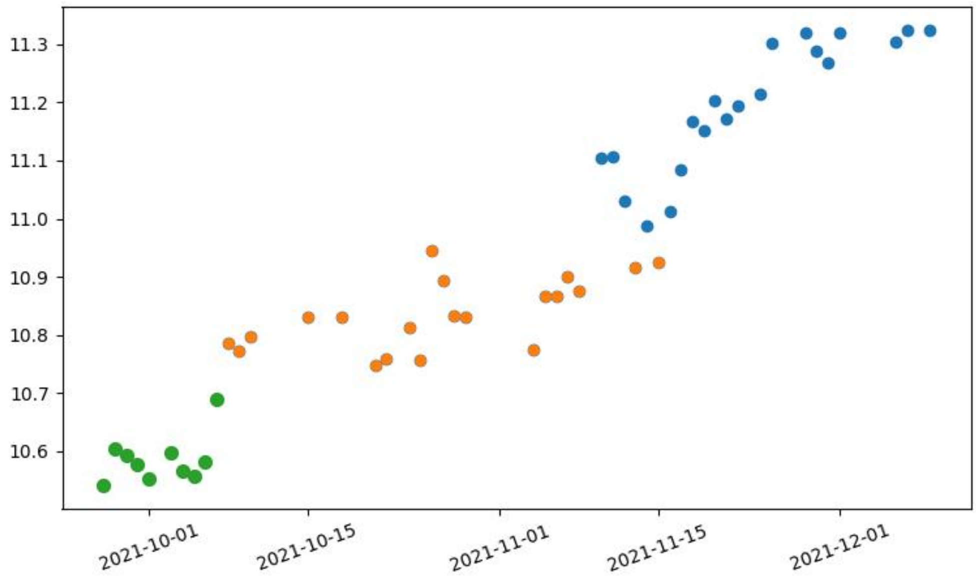

Figure 7.8: Card 12: best historical prices

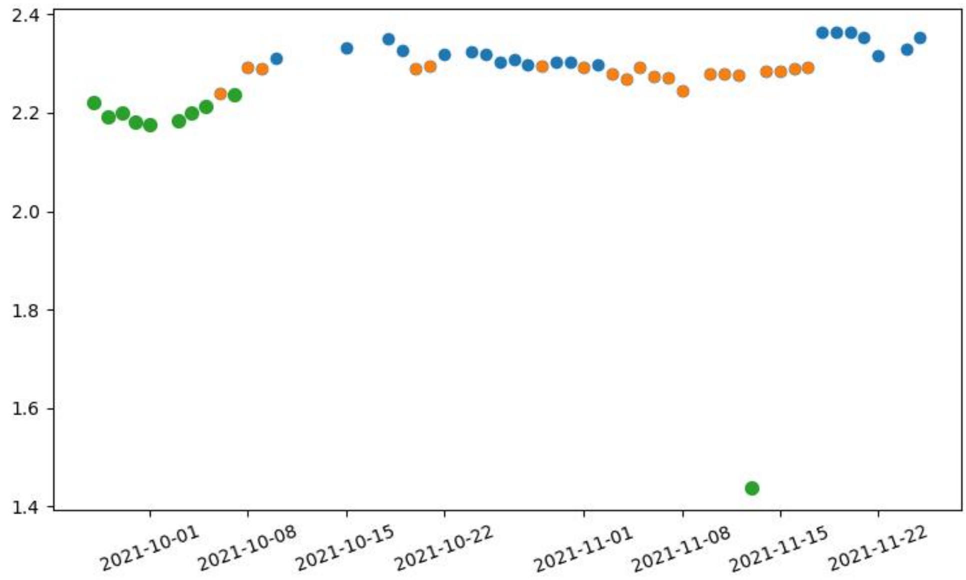

Figure 7.9: Card 21: best historical prices

but not in the database, which results in an update by the manager. The data is successfully isolated, then decompressed and inserted into tables. The whole process takes about 10 minutes, and ends in saving the new checksum and cleaning up temporary local files (Fig. 7.13). On the same figure we see the data mining module finally getting to perform the analysis, as the database checksum is the same as the last checksum of the files. Database connection is established, helper tables are generated and the data is read and processed into charts, completing the data pipeline.

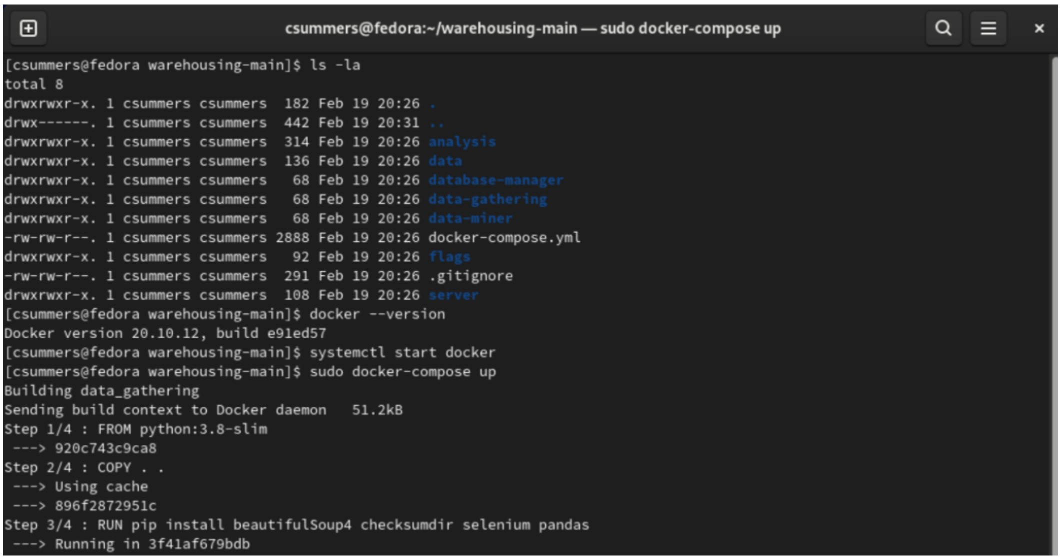

Figure 7.10: Fedora — running the system

Overall, the system proved to be resistant to environment changes — given the *Docker* infrastructure is available — which is one of the main goals of containerization.

7.3 Testing

The project was being developed without writing unit tests. Proper execution of the code was everytime assesed in conditions with some data, full data and none date. For minor changes and in initial stages, the containers were booted up separately, but with the transition to multi-container application, the internal specification of each container became dependent of the system they create with other containers (e.g. the webdriver container is visible from within the inner network, and a few elements work without the MySQL database container). All the needed testing has been done manually, but most importantly, the correctness of the results of given operation was the primary indicator of the

```
Creating mysql_database    ... done
Creating firefox_webdriver ... done
Creating sematext_agent    ... done
Creating data_miner        ... done
Creating node_server       ... done
Creating data_gathering    ... done
Creating database_manager  ... done
Attaching to node_server, data_gathering, data_miner, database_manager
node_server            | Listening on 8080
data_gathering         | Started: data-gathering
database_manager       | Started: database-manager
node_server            | Database connection established.
data_miner             | Started: data-miner
data_gathering         | Date ID [135] already added.
database_manager       | Local files checksum: f0aabdef2532629957ca159898b8bdb62764bb4e
database_manager       | Database checksum: 440d1504250e10388a24b179ded9d29c2c329960
database_manager       |  - New data found, but is not complete. Waiting 15 minutes.
data_gathering         |  - Checking for completeness of local files.
data_gathering         |    - Data validation completed successfully.
data_gathering         |    - Saving checksum: f0aabdef2532629957ca159898b8bdb62764bb4e
data_gathering         |  - Job is done. Waiting for 1 hour.
data_miner             |  - New data found to be loaded into the database. Waiting 5 minutes.
```

Figure 7.11: Fedora — containers behaviour on system init

```
data_gathering         |  - Checking for completeness of local files.
data_gathering         |    - Dataset already validated. All needed data saved.
data_gathering         |  - Job is done. Waiting for 1 hour.
database_manager       | Local files checksum: f0aabdef2532629957ca159898b8bdb62764bb4e
database_manager       | Database checksum: 440d1504250e10388a24b179ded9d29c2c329960
database_manager       |  - Verified new data available for database update.
database_manager       | Data isolated.
database_manager       | Database connection established.
database_manager       | Decompressing seller.csv...
database_manager       | Decompressing sale_offer.csv...
data_miner             |  - New data found to be loaded into the database. Waiting 5 minutes.
database_manager       | Decompressing date.csv...
database_manager       | Decompressing card_stats.csv...
database_manager       | Decompressing card.csv...
database_manager       | Updating seller...
database_manager       | Updated seller in 2.408 seconds.
database_manager       |
database_manager       | Updating sale_offer...
node_server            | Date ID: null.
data_miner             |  - New data found to be loaded into the database. Waiting 5 minutes.
data_miner             |  - New data found to be loaded into the database. Waiting 5 minutes.
database_manager       | Updated sale_offer in 607.528 seconds.
database_manager       |
database_manager       | Updating date...
database_manager       | Updated date in 1.455 seconds.
database_manager       |
database_manager       | Updating card_stats...
database_manager       | Updated card_stats in 3.113 seconds.
database_manager       |
database_manager       | Updating card...
database_manager       | Updated card in 0.656 seconds.
```

Figure 7.12: Fedora — database update

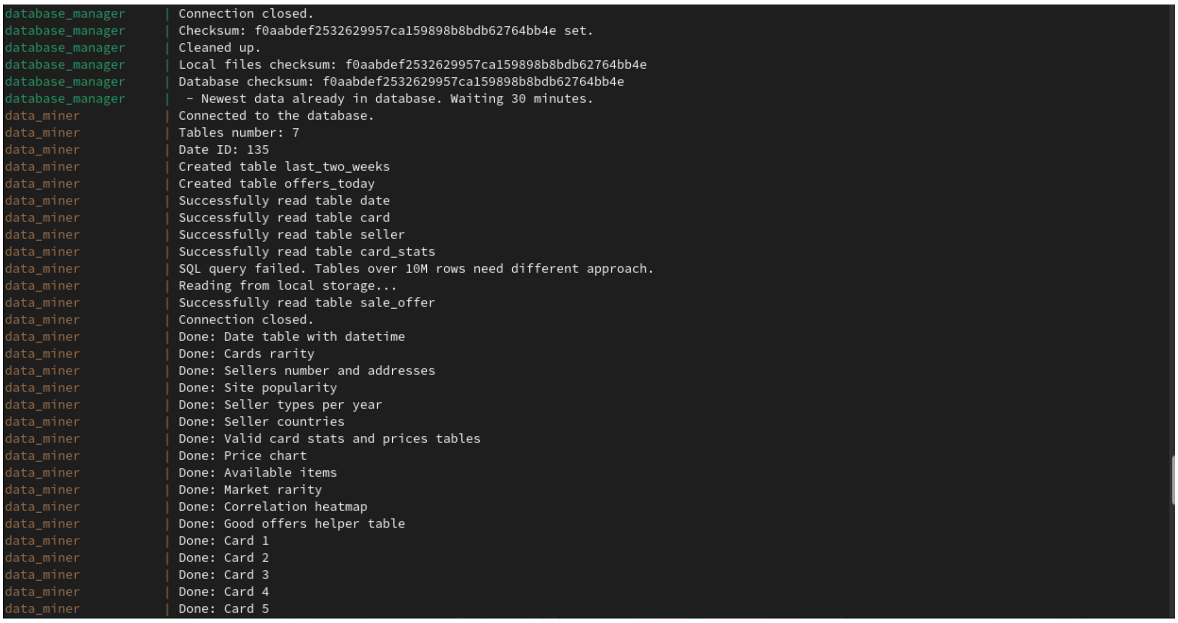

Figure 7.13: Fedora — data mining and analysis

code stability. The application was also built with the intent of running on different operating systems without problems and extensive setup. This is possible due to containerization, which provides an own microsystem for each application, isolating the core of the program from the host system.

The program is able to behave accordingly in unexpected conditions, as presented on Fig. 7.14. On encountering an exception, the pickled data is first retrieved, then the container is set up to restart in 30 minutes. Another example is finding a sale offer put by a user, that hasn't been saved in the sellers table, which marks the dataset as not valid (Fig. 7.15).

7.4 Version control and project variation in time

Initially, during the *DWDMS* course, the project was a python program composed of several python scripts that the author and his colleagues used to gather and store the card market data. A data analysis GUI module has been implemented to showcase the possibility of utilizing the collected data. Then, the transformation of the author's part into containerized version happened, with more verbose logging. All changes were tracked using *git* version control system integrated with Visual Studio Code environment. Over the span of months, the implementation of the data warehouse underwent major changes with

```
Started: data-gathering
Date ID [54] already added.
 - Force update flag active. Proceeding to run.
Webdriver connection ready

Loaded card: 264 rows
Loaded card_stats: 14179 rows
Loaded sale_offer: 4543650 rows  [82941 rows selected, 98.17% reduced]
Exception occured while loading date.pkl

[Errno 2] No such file or directory: './.pickles/date.pkl'
"['id'] not found in axis"
 - Unpickling data. Container will restart in 30 minutes.
Merging sale offers: -82941, +82941 rows
Size before: 4543650
Size after: 4543650
```

Figure 7.14: Handling an exception during the initial stages of gathering run

```
Started: data-gathering
Date ID [54] already added.
 - Checking for completeness of local files.
Seller from sale offer not saved in sellers
   - Gathered data is incomplete. Proceeding to run.
Webdriver connection ready

Loaded card: 264 rows
Loaded card_stats: 14179 rows
Loaded sale_offer: 4543650 rows  [82941 rows selected, 98.17% reduced]
Local pickle data validated (removed 0 records)

Task - Getting all card names from current expansion
URL change  ->  https://www.cardmarket.com/en/Magic/Products/Singles/Battlebond?site=1
Done - All card names from Battlebond saved

 == Skyshroud Claim ==    (1/264  0.38%)
URL change  ->  https://www.cardmarket.com/en/Magic/Products/Singles/Battlebond/Skyshroud-Claim
                Expanding page...
                Time: 7.366

 = Card stats =
Card ID:       1
Price from:    0.05
Averages:      1.39, 1.11, 1.05
Amount:        1120
Date ID:       54

 = Sellers =
Task - Updating sellers
Seller:  NikkiFrikkel from Germany [7280]
Done - 1 new sellers saved  (out of: 157, total: 7280)
Time: 2.725
```

Figure 7.15: Handling invalid datasets — running to complete a list of sellers wrt. sale offers

318 commits sent to the project. Changes in the website had to be appropriately mirrored in the gathering part of the code, time-based querying was swapped for Selenium explicit and implicit waiting methods, pickle format was added and many different SQL connection engines were tried to combat side effects of large datasets. The code was meticulously improved and polished, until the gathering time was decreased from 2.5+ hours to 1–2 hours, with most of the runs finishing in about 75 minutes. Some challenges that were faced are still to be solved, for example solving the seller dictionary fiasco when building the object iteratively and trying to save it in *.pkl* format intermediately. Another issue is loading a full sale offers table into a dataframe, when it has over 10 million rows, which takes a long time and often results in a out-of-memory error. Keeping the best programming pracices in mind, all remaining encountered problems were solved, so that the system would perform its tasks and fulfill the requirements stated in sections 3.1 and 3.2.

Chapter 8

Summary

The presented thesis revolved around managing multiple Docker containers, each responsible for a part of the data pipeline, gathering and transforming card market information from a public website into a user-oriented analysis. The constraints and requirements described in sections 3.1 and 3.2 shaped the system to perform a series of robust tasks, in order to aid the potential buyer with novel market insights. In retrospect, the program does fulfill the following functional requirements:

1. Gathering cards, sellers and offers' data and keeping it up to date — realized by the *data_gathering* container, mostly with the use of scheduling flags and Selenium framework,
2. Possibility to change the expansion — it is a configurable variable in *config.py* file of the gathering container,
3. Saving the gathered entities into CSV files — staging area,
4. Continuous execution of every container,
5. Once-per-day data update,
6. Once-per-hour database update checks,
7. Once-per-hour data mining run,
8. Having a robust web application with four data analyses.

Non-functional requirements were concerned with *how* does the system behave, rather that *what* it does. All of them have been fulfilled:

1. Full data processing time is about 1.5-2.5 hours, given that all the sellers are saved. This requirement fails for the first run of the code, especially if the Internet connection and hardware setup are suboptimal.

2. The system is standalone within the containerization software, the only prerequisite to running it is having Docker installed on the system. Has been tested on Fedora 35 with minimal setup to a triumphant success.

3. The requests slow down after no proper response is detected, which is not dynamically chaning the requests timing, but rather giving the server a break to avoid running into error 429 Too Many Requests.

4. No user input is required after running the system in order to keep it healthy, functional and up to date. The only interaction the user has is receiving the results and putting their conditions to find out what the best deal is.

Overall, six containers were used, four of which were sub-projects acting on the data (data gathering, database managing, data mining, website serving). Additionally, a Sematext monitoring software was used to observe the performance of the system, which amounted to additional *sematext_agent* container, reporting execution details to the dashboard.

8.1 Containers performance

Using the free version of Sematext monitoring agent, the author was able to track the performance of the system, mainly the MySQL database.

In Fig. 8.1 we can see the amount of data received and transmitted by the database. The steady hourly level represents the data mining module, which connects to the database every 60 minutes to perform the analysis. Spikes around 1:00 am correspond to the database manager updating the tables with new data gathered from this day.

Fig. 8.2 shows the load of the database. We can see the correlation between the Selects Rate and CPU and Memory load. Around 4 GB of memory are used, with additional 2–3 GB chached and buffered.

The last figure juxtaposes the number of tables dropped with tables created (Fig. 8.3). Due to the architecture of the system, each table is replaced by its new version, meaning every CREATE is paired with a DROP, which is reflected in the presented chart.

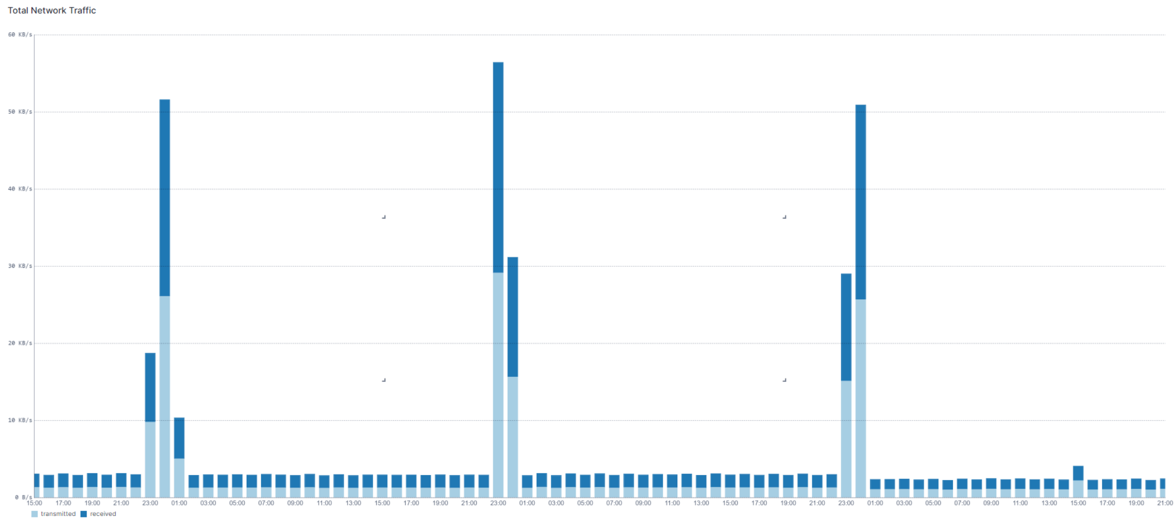

Figure 8.1

8.2 Scaling and customizability

The system was build with the idea of scalability from the very beginning. The number of gathered data can be extended to years, as long as the largest table can be loaded into a single dataframe without running out of memory. Interestingly, if the data gathering part was swapped with other code for data acquisition, the data warehouse could instead be focused around the new theme and still work with few minor changes to the other containers. One may want to choose another website, and if the information provided there is sufficient to create the same entities in CSV files, there would be no changes needed for the rest of the data pipeline to work correctly. Moreover, the usage of containers (which can be asynchonously stopped and started) suggests a version of the system with multiple versions of each container, each still doing their core task, but with differences. Such modular approach is seen more and more in the world of mordern technology and would be perfectly applicable here. Additionally, the acquired data can be used in a myriad of different ways besides those presented in this paper. The created data warehouse is simple, but proves to be a powerful tool, which usefulness can be extended far with the right approach and broad imagination.

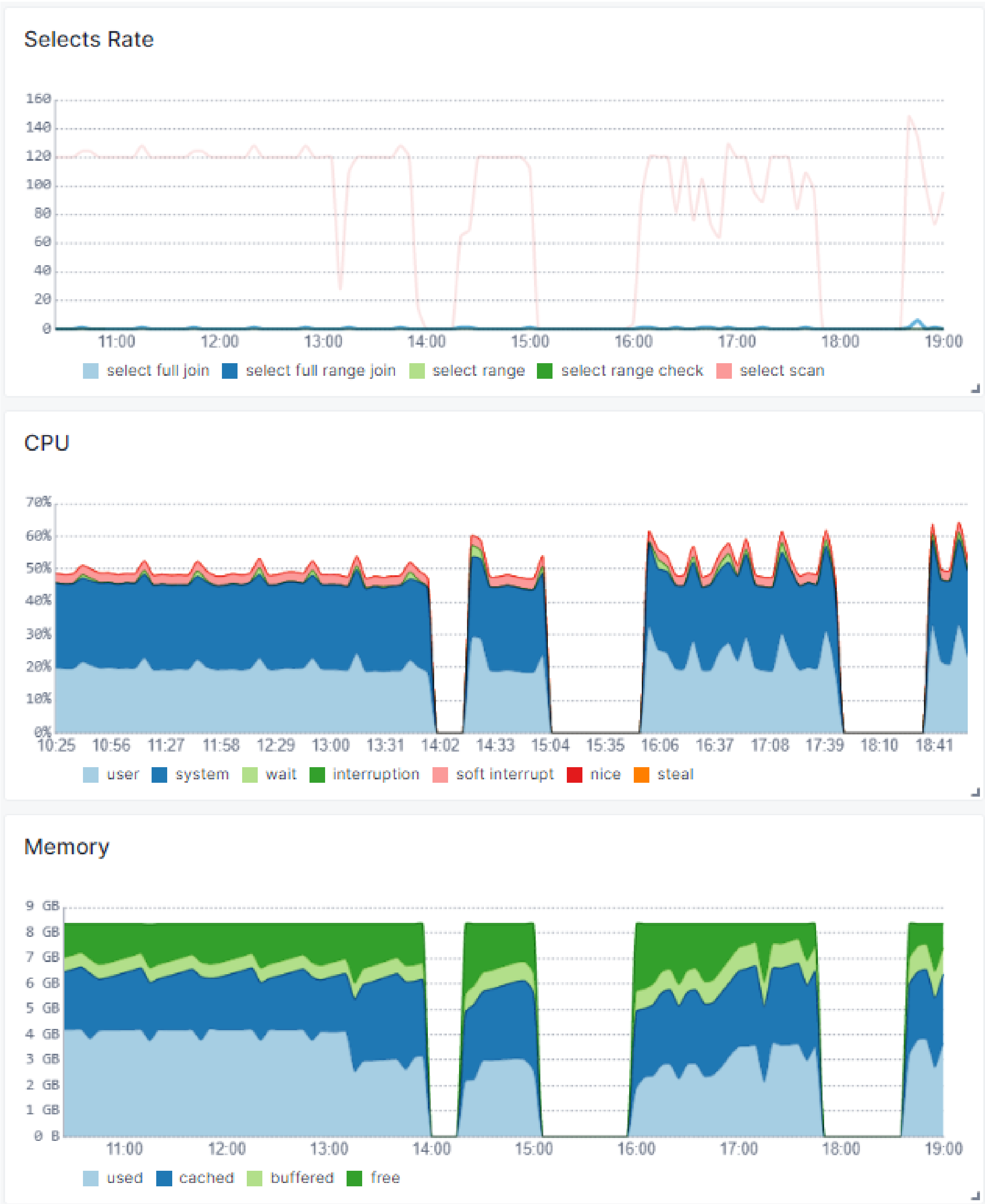
Selects Rate
160
140
120
100
80
60
40
20
0
11:00
12:00
13:00
14:00
15:00
16:00
17:00
18:00
19:00
select full join
select full range join
select range
select range check
select scan
CPU
70%
60%
50%
40%
30%
20%
10%
0%
10:25
10:56
11:27
11:58
12:29
13:00
13:31
14:02
14:33
15:04
15:35
16:06
16:37
17:08
17:39
18:10
18:41
user
system
wait
interruption
soft interrupt
nice
steal
Memory
9 GB
8 GB
7 GB
6 GB
5 GB
4 GB
3 GB
2 GB
1 GB
0 B
11:00
12:00
13:00
14:00
15:00
16:00
17:00
18:00
19:00
used
cached
buffered
free

Figure 8.2

Figure 8.3

8.3 Conclusions

In conclusion, the created project is a successfully working solution. It performes the totality of given tasks, works relatively fast, uses verbose logging and it set up to be resilient and easy to use. The data warehouse model fits well with the purpose of the program and containers allow to isolate the steps on the data pipeline, as well as transport the solution to any system and run it there. The results are calculated fast and often, and the web application provides answers to questions, which can substantially contribute to the user making a better market choice. Using the available technology the author was able to transform large, disseminated pieces of information into centralized and uniform data, to then extract insights in a matter of seconds, whereas the market research would have taken days, if not weeks, spent on browing the service websites. This sufficiently proves the real usefulness of today's technology and the advantage in hands of those who use it well. However, if data is the sword of the 21st century, one must always be cautious not to get cut; issues with big data are challanges many of the biggest companies face today, and the ethics of discovering knowledge about people, which even they do not have, opens a whole new Pandora's box of moral dilemmas. Just like the technology promising improvement in our lifestyles, data is a double-edged sword. It's giving us an advantage, but is also placing a great burden of responsibility — a burden, which has to be carried to the end once the lid is opened.

Bibliography

[1] Barry A. Devlin and Paul T. Murphy. “An Architecture for a Business and Information System”. In: *IBM Syst. J.* 27 (1988), pp. 60–80.

[2] William H. Inmon. “The data warehouse and data mining”. In: *Communications of the ACM* 39 (11 1996). URL: https://link.gale.com/apps/doc/A18993844/AONE?u=anon~8d447f8d&sid=googleScholar&xid=9f3e529f.

[3] Suresh Kotha. “Wizards of the Coast”. Archived from the original (PDF) on September 1, 2006, retrieved August 11, 2013. Oct. 1998. URL: https://web.archive.org/web/20060901100217/http://faculty.bschool.washington.edu/skotha/website/cases%20pdf/Wizards%20of%20the%20coast%201.4.pdf.

[4] Guido Van Rossum. “The History of Python: A Brief Timeline of Python”. Archived from the original on 5 June 2020. Retrieved 5 March 2021. Jan. 2009. URL: https://python-history.blogspot.com/2009/01/brief-timeline-of-python.html.