FACULTY OF INFORMATION TECHNOLOGY CTU IN PRAGUE

ASSIGNMENT OF BACHELOR’S THESIS

Title:	Sensitive Data Detector for Databases
Student:	Viktor Dohnal
Supervisor:	Ing. David Knap
Study Programme:	Informatics
Study Branch:	Computer Security and Information technology
Department:	Department of Computer Systems
Validity:	Until the end of summer semester 2020/21

Instructions

The goal is to design a crawler detector capable of pattern-based detection and scoring of data sensitivity in an unknown database.

1. Explore possible ways of automated crawling of unknown databases for major database management systems (MySQL/MariaDB, PostgreSQL, Oracle DBMS).
2. Define search patterns for sensitive data detection (based for example on the General Data Protection Regulation and Telecommunications Act of Czech Republic).
3. Design architecture of a sensitive data detector in unknown databases focused on configurability of data patterns.
4. Perform a prototype implementation of the designed architecture and perform tests of its ability to find and score sensitive data in an unknown database.

References

Will be provided by the supervisor.

prof. Ing. Pavel Tvrdík, CSc.
Head of Department

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
Dean

Prague October 22, 2019

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Bachelor's thesis

Sensitive Data Detector for Databases

Viktor Dohnal

Department of Computer Systems
Supervisor: Ing. David Knap

June 4, 2020

Acknowledgements

I would like to express my gratitude to my supervisor, Ing. David Knap, for his guidance, insights, and the valuable feedback he provided me with.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as school work under the provisions of Article 60(1) of the Act.

In Prague on June 4, 2020 . . . .

Czech Technical University in Prague
Faculty of Information Technology
© 2020 Viktor Dohnal. All rights reserved.
This thesis is school work as defined by Copyright Act of the Czech Republic. It has been submitted at Czech Technical University in Prague, Faculty of Information Technology. The thesis is protected by the Copyright Act and its usage without author's permission is prohibited (with exceptions defined by the Copyright Act).

Citation of this thesis

Dohnal, Viktor. *Sensitive Data Detector for Databases.* Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2020.

Abstrakt

Organizace po celém světě pracují se stále větším množstvím citlivých dat a dá se očekávat, že tomu tak bude i nadále. Aby nedošlo k jejich úniku či zneužití, je důležité provádět pravidelné audity a úložiště citlivých dat mapovat. Jelikož je audit velké podnikové databáze časově velmi náročný, je žádoucí mít k dispozici nástroj, který je schopný taková data v databázi detekovat automaticky. V této práci je nejprve krátce zanalyzována legislativa, která určuje, co lze za citlivá data považovat. Následně je diskutován výběr vhodných technologií k rozpoznávání těchto dat a vypracován návrh aplikace, která bude detekci provádět na neznámé databázi, přičemž jedním z hlavních cílů je její modularita. Nakonec je tento návrh implementován a otestován na dostupných datech.

Klíčová slova databáze, telekomunikace, crawling, citlivá data, audit databází, regulární výrazy, klasifikace dat, Intel Hyperscan

Abstract

The amount of sensitive data stored by organizations throughout the world is increasing, and this trend is likely to continue. To prevent data breaches and misuse, it is necessary to maintain records of sensitive data storage and conduct audits periodically. Given the fact that auditing a huge enterprise database is very time-consuming, there is a need for automatic detection of such data. The first part of this thesis is devoted to determining the definition of sensitive data in the current legal environment. These definitions are then used to select proper technologies and to design an application that is able to detect this data in an unknown database where one of the main objectives is modularity. Finally, this design is implemented and tested on available datasets.

Keywords database, telecommunications, crawling, sensitive data, database auditing, regular expressions, data classification, Intel Hyperscan

Contents

List of Figures

Introduction

Over the past few years, we have grown more and more dependent on modern technology and our habits have changed widely. Social media, online shopping, location-based services, and much more have created a lasting trail of our habits and preferences. While some of this may still be influenced by one's own behavior, there are many cases where opting out is not a possibility.

With an ever-growing amount of personal data being stored throughout the world, organizations in charge of storage can quickly become overwhelmed. This can lead to unintentional data leakage or even theft by cyber adversaries. Since data breaches can affect nearly everyone, potentially involving extreme cases such as identity theft, regulatory attempts have been made to address the situation. With older, country-specific regulations for the telecommunication industry or more recent ones, such as the General Data Protection Regulation, companies are legally obligated to handle data accordingly.

To protect something (and therefore comply with these regulations), one first must know where it is. For a larger organization, this may present a tough challenge. As applications are being actively developed and infrastructure maintained with each release cycle, update or change, personal data can easily remain forgotten, waiting for a disaster to happen. In the telecommunications industry, selected as the main field of interest for this thesis, one organization can have as much as hundreds of different databases, that require maintenance. The stakes here are high, as making a mistake can lead to enormous fines as well as loss of customer trust.

One of the critical steps to keeping sensitive data well maintained and protected is periodic auditing by designated company auditors. Of course, auditing hundreds of large databases manually is nearly impossible, so there is a need for tools that can help face these problems. The main goal of this thesis is to design an application that could serve as a tool for auditing in such environments.

The first part of this thesis explores the possibilities of sensitive data discovery in such databases with a focus on regulatory requirements and a brief look at the commercial tools available. Afterward, it focuses on the custom solution. The goal of this thesis is to design an application that could serve as a tool for auditing in the previously mentioned environments.

Goals

The goals of this thesis are as follows.

- **Define sensitive and personal data**
  Based on the current regulatory environment for the telecommunications industry in the Czech Republic, define what records precisely (if possible) can be considered personal/sensitive data.
- **Explore ways of crawling a database**
  Find suitable algorithms and processes used to crawl an unknown database.
- **Specify search patterns**
  Based on the definition of personal/sensitive data obtained previously, specify search patterns to detect such data.
- **Select appropriate technologies**
  Consider previous facts for choosing the most appropriate technologies for the application. Selected technologies should have a license compatible with the nature of the project, so at least unrestricted commercial use should be allowed.
- **Design application architecture**
  Design an application with a focus on modularity (users can add new patterns easily) and performance.
- **Implement the design**
  Create a prototype implementation of said application. It should be cross-platform, with at least Microsoft Windows, Apple macOS, and GNU/Linux operating systems supported.
- **Evaluate performance**
  Test applications ability to detect sensitive/personal data in an unknown database and create performance benchmarks. Evaluate both modularity (how hard is it to add support for a new database or add a new pattern) and the end-user experience.

Chapter 1

Information Privacy Law

In the past two decades, data breaches have become a relatively common occurrence. One could argue that it is already beyond what an average person can keep track of. According to [3], in 2018, there have been over 1,200 reported data breaches in the United States alone, exposing a total of 446 million records. It is essential to emphasize the word "reported" - there may very well be many more cases. The organizations affected by these data breaches span widely across many different industries, as well as the list of methods used in the attacks. Be it healthcare, business, education, or military; no industry seems entirely safe from insider theft, hacking, employee negligence, or even accidental internet exposure. For consumers, these statistics are alarming since having one's data exposed can be potentially devastating.

A good example would be the 2017 Equifax data breach. As described in [7], Equifax, one of the three largest consumer credit reporting agencies in the United States, lost the personal information of about 147 million people. The lost data contained names, addresses, dates of birth, Social Security numbers, drivers' license numbers, and even credit card numbers. While this certainly did not help customer trust, it also had more direct implications for the company. As of July 22, 2019, Equifax had agreed to a global settlement, which included up to $700 million of which $425 million would be used directly to help people affected by the data breach. [10]

And such financial damage to organizations is not uncommon. According to [2], the average total cost of a data breach is nearly $4 million, with healthcare being by far the most expensive industry to date.

It is then no wonder that governments all across the world try to limit the harm caused by irresponsible data management. The idea that the manipulation with personal data should be regulated goes back to the late 1960s, long before widespread internet access for most of the Western population. On 11 May 1973, Sweden's Data Act was enacted as the world's first comprehensive national data protection law [14]. Since then, the legislation regarding data protection has gone through a long journey. In January 2017, there were a total of 120 national data privacy laws in place worldwide with more on the way [13]. These laws create pressure on the organizations to handle data accordingly, specifying the financial penalties incurred if they fail to do so.

In this thesis, the main focus is on the law regulating the Czech telecommunication industry from the personal data perspective. Currently, there are two such laws in place: the Telecommunications Act of the Czech Republic and the Czech implementation of the General Data Protection Regulation (GDPR). Each of these is being discussed separately in the upcoming sections.

1.1 Telecommunications Act of the Czech Republic

The Telecommunications Act (127/2005 Sb.) [30] [1] was enacted in March 2005, and since then, amended several times. Like its predecessor (151/2000 Sb., as of now repealed), it is a comprehensive act regulating the whole telecommunications industry in the Czech Republic. It specifies the Czech Telecommunication Office as its primary enforcer. It also lists various responsibilities for the office, such as the assignment of cellular frequency bands and their monitoring, maintenance of phone number databases, cooperation with the corresponding government departments on new law proposals, and customer protection.

Rules for data protection and governance are mainly defined in chapter 1 of title 5, with corresponding violations and fees defined in § 118 of title 7. In § 87, it is stated that all rights and duties regarding personal data protection not regulated in this section are governed by special (designated) regulations, therefore **from the perspective of data protection, subjects in the telecommunications industry are regulated like any other subjects unless stated otherwise.** Up until 2018, this indirectly referred to the Data Protection Act (101/2000 Sb.), which was then repealed and replaced by the GDPR, both of which are discussed below.

Governance of **data specific to telecommunications, such as messages, phone calls, and location data, are regulated directly by this act**, as specified in § 88, § 88a, and § 89. As this data is (arguably) of extraordinary value, these sections specify that their storage must be well secured as well as what actions the organization must take if it fails to do so. § 97 states that such data must be stored only for a period of six months (for the needs of law enforcement organizations) or if it is necessary for the correct operation of the service (operational data). If none of the previous applies, data must be either anonymized (see definition of anonymous data below) or disposed of. Should the organization fail to protect the data or mishandle them, according

[1] Information regarding this and other Czech acts were translated by the author of this thesis since no English translation was available

to § 118, it can be fined up to 50 000 000 CZK or 10 percent of its annual revenue of the prior financial year, whichever is higher.

Data Protection Act (101/2000 Sb.) [28] was enacted in April 2000. While at the time of enactment, only about 10 percent of Czech individuals had access to the internet [24], this act shares many features with its 16 year younger successor, GDPR. The act applied to all data processing except household or purely personal activities.

§ 4 specifies three types of data;

- personal data,
- sensitive data,
- and anonymous data.

Personal data is any piece of information related to an identified or identifiable data subject. The data subject is considered identified or identifiable if it can be identified directly or indirectly based on a number, code or one or more factors, specific to its physical, physiological, mental, economic, cultural or social identity. **Sensitive data**, being a subset of personal data, is any piece of information related to national, racial or ethical origin, political views, trade union membership, religion or beliefs, criminal convictions and offenses, medical status, genetic status, and sexual orientation. Finally, anonymous data is any data that, either in its original form or processed, cannot be matched with any identified or identifiable data subject.

As one would expect, when storing data from the sensitive category, more strict rules apply than when storing data purely from the personal category. The fines for mishandling the data are also different.

The act was repealed in April 2019 and **replaced with Personal Data Processing Act (110/2019 Sb.) and the GDPR.**

1.2 General Data Protection Regulation

Regulation (EU) 2016/679 of the European Parliament and of the Council of 27 April 2016 on the protection of natural persons with regard to the processing of personal data and on the free movement of such data, and repealing Directive 95/46/EC [22], in short, referred to as General Data Protection Regulation or GDPR, is a comprehensive data protection regulation applied in all member countries of the European Union and European Economic Area. It came into force in May 2018. Some national states also introduced their specification of the regulation to complement the GDPR as standalone national acts. These acts can add small (permitted) modifications and specifications of the regulation for the given country. Most of the time, these closely match the GDPR and differ only in things like the age of consent.

In the case of the Czech Republic, the act in question is the Personal Data Processing Act (110/2019) [29]. This act, for example, establishes the Czech Telecommunication Office as its enforcer and specifies exemptions from the regulation for use cases like academia. Since it does not introduce any new definitions regarding sensitive or personal data in the private sector, this act is not addressed further in this thesis.

In article 4 (definitions), GDPR defines these subsets of data;

- personal data,
- genetic data,
- biometric data,
- and data concerning health.

The terms genetic data, biometric data, and data concerning health are used later in article 9. **Genetic data** is *"personal data relating to the inherited or acquired genetic characteristics of a natural person which give unique information about the physiology or the health of that natural person and which result, in particular, from an analysis of a biological sample from the natural person in question"*, **biometric data** is defined as *"personal data resulting from specific technical processing relating to the physical, physiological or behavioural characteristics of a natural person, which allow or confirm the unique identification of that natural person, such as facial images or dactyloscopic data"* and **data concerning health** is defined as *"personal data related to the physical or mental health of a natural person, including the provision of health care services, which reveal information about his or her health status."*

Personal data is defined as *"any information relating to an identified or identifiable natural person ('data subject'); an identifiable natural person is one who can be identified, directly or indirectly, in particular by reference to an identifier such as a name, an identification number, location data, an online identifier or to one or more factors specific to the physical, physiological, genetic, mental, economic, cultural or social identity of that natural person."* Note that apart from the genetic and biometric identity, the definition is very similar to the previously mentioned Data Protection Act, which is nearly 20 years old.

Although not explicitly defined, article 9 indirectly specifies another set of personal data with more restrictive rules for processing and storage. This data is defined as *"personal data revealing racial or ethnic origin, political opinions, religious or philosophical beliefs, or trade union membership, and the processing of genetic data, biometric data for the purpose of uniquely identifying a natural person, data concerning health or data concerning a natural person's sex life or sexual orientation."* This is also nearly identical to the definition of the sensitive data from the Data Protection Act. Note, however, that data about criminal convictions and offenses are not included. This is because, in the GDPR, it is defined separately in article 10, which is rather clear about who can store such data: *"Any comprehensive register of criminal convictions shall be kept only under the control of official authority."* For the needs of this thesis, this is also included in the same data group.

When it comes to scoring the data based on sensitivity, the situation gets more tricky. This is because fines for mishandling the data are not specified in detail for each of these defined subsets. For most of the violations, in theory, the organization can be fined *"up to 20 000 000 EUR, or in the case of an undertaking, up to 4 % of the total worldwide annual turnover of the preceding financial year, whichever is higher."* Using imminent financial losses as a factor when scoring is therefore difficult, if not impossible, and definitely beyond the scope of this thesis.

1.3 Implications for Sensitive Data Patterns

As was discussed in the previous chapter, scoring sensitivity of the data based on potential financial losses incurred, should it, for example, leak, is not an option. GDPR, however, provides one clear distinction, and that is the difference between personal data and sensitive data (as defined in article 9). Since manipulation with sensitive data is a lot more restrictive, it can be argued that its leakage could be more severe, and therefore worse from the financial standpoint. Data from the previous sections can then be divided into the following groups based on their type and sensitivity.

- **Data Specific to Telecommunications**
  The Telecommunications Act directly regulates manipulation of data from this category. This data includes
  - messages (SMS, MMS) and metadata about them (e.g., time sent or time received),
  - phone calls and metadata about them (who called whom and when, for how long, etc.),
  - internet usage (e.g., traffic logs or an IP address),
  - and location data.
- **Personal Data**
  This data is currently defined by the GDPR (see above). Since its definition is so broad, there is no definitive list of what it can include. It certainly includes
  - first and last name (and maiden name),
  - date of birth,

  - any government-issued ID like a passport or a drivers' license,
  - Social Security number or any national identification number,
  - email address (non-personal address like info@company.xyz is most likely not included),
  - physical address or any part of it (city, zip code, etc.),
  - phone number,
  - credentials like username or password,
  - banking information like a personal account number or credit card number,
  - financial information like an account statement, loan records or property records
  - web-related information like browser cookies, search and browsing history or user ratings

  It is important to note that this data has to relate to a person - a company phone number does not require the same protection level as a personal phone number. As stated before, the list presented is by no means exhaustive.

- **Sensitive Personal Data**
  This is a subset of personal data defined by the GDPR (articles 9 and 10, see above). While its definition is considerably more narrow, there still is not a complete list available. In more general terms, it includes

  - racial, national or ethnic origin,
  - political opinions,
  - religious or philosophical beliefs,
  - genetic data,
  - biometric data (e.g., a fingerprint or a face),
  - data concerning health,
  - data concerning a natural person's sex life or sexual orientation,
  - and data about criminal convictions and offenses.

  Definitions of genetic data, biometric data, and data concerning health can be seen above.

These groups are not disjunct. Data specific to telecommunications can also fall into the personal data definition from the GDPR, and sensitive data is a subset of personal data. The primary purpose of these groups is to create some level of distinction (i.e., scoring) for the user of the application. In general, leaking personal email addresses is less damaging than leaking medical data (at least according to the GDPR), and the application should present the difference in severity accordingly.

When examining these data definitions, it is clear that some of them may be easier to detect than others. Social Security numbers, email addresses, or phone numbers have standardized forms and generally can be expected to look the same and be handled in the same way. This gets a bit harder when it comes to names or physical addresses. These should generally consist of some known values, like all Czech first names or every possible city in the state of Texas, but it highly depends on the cultural environment. For most of them, it should be possible to tell, with some level of certainty, if the one-word string in question is the first name or not. The last category of this hypothetical division is data with no standard way of storage, like data concerning health or criminal convictions. These can be stored in a lot of different ways, from the weight as a number to a detailed description of the patient's response to experimental treatment. Most of the sensitive data (as described above), falls within this category.

Let's give these categories a more formal definition so they may be incorporated into the design of the application. These categories are also used later for selecting the proper technologies.

- **Defined Format**
  A piece of information that has one or a couple of formats in which it can appear. It can be standardized, like an RFC 5322 email address, or without an official standard, but expected to be stored in a similar fashion, like a location.

- **List of Known Values**
  Data that is expected to contain one or more defined substrings. The obvious example is name or surname, a list of which can be obtained for a specific country or nation. Another example would be a list of known professions. It does not always have to produce a match, but more often than not, it should.

- **Unknown Format and Unknown Values**
  Information that could be stored in many ways and contain many different values. It has no standard nor can be reasonably expected to look similar across different databases and use cases. A political opinion could be a set of numbers as a result of a survey, coordinates on a political compass, or a political party preference. Much the same could be said for trade union memberships. Sensitive data (as described above), usually fall within this category.

Chapter 2

Data Protection

Laws and regulations compliance is only a mere subset of the data security field. The last decades of rapid development and research have lead to new methodologies, terms, and even dedicated job positions. For most systems, hypothetically, privacy is non-essential. Sure, loss of customer trust, legal obligations, or ethical concerns are a factor, but in theory, the absence of privacy by itself is not what causes the system to stop working. **Privacy engineering** is defined in [6] as *"the discipline of understanding how to include privacy as a non-functional requirement in systems engineering."* It focuses on risk models, privacy methodologies (privacy by policy, privacy by architecture and technical point control), and even non-technical considerations of privacy, such as how cultural and social norms affect it.

An example of privacy by policy would be laws and regulations compliance mentioned at the beginning. In practice, this usually means gathering all information about the regulatory environment the system will operate in and then incorporating these requirements into the design. The goal of privacy by architecture is to *"provide a solution that performs the business function that the system was built for and does so in a privacy preserving manner."* [6] One of the measures of privacy by architecture is **data anonymization**. In [27], it is defined as *"the process of de-identifying sensitive data while preserving its format and data type."* There are many reasons for using data anonymization, one of which is legal - many privacy regulations require it in some form (see the previous chapter). Another exemplary use case is a separation of testing and production environments. Testers or contractors can then use the anonymized version of the data to prevent leakage. Before any anonymization can take place, one must know where the sensitive data is. While some anonymization tools do this automatically, others require users to specify the data themselves. Automatic detection arguably is not of the highest importance since all the factors (and therefore, data inputs) are known in the design phase of the project—the decision of what data should be anonymized, and how, can be decided at the drawing board.

The need for automatic detection arises in the third group of methodologies,i.e. technical point control. These methodologies attempt to reduce privacy risks that cannot be mitigated by the architectural approach or were not considered during the design phase. **Data loss prevention (DLP)** is a comprehensive approach that identifies and protects data at rest, data in use, and data in motion. [37] For these three kinds of data, DLP has corresponding objectives. Keep in mind that this applies to all forms of data, not just databases. Data at rest, for example, a database or a Word document in cloud storage, should be located and cataloged. An organization should know at all times where its data is and what information it comprises of. Data in motion is any data that is being transferred from, to, or on the organization's network. This includes data like emails, instant messages, or web browsing. DLP systems should prevent or alert responsible personnel in case of any unauthorized transfer of sensitive data. Note that sensitive data can include information outside the regulatory definition - company know-how, while not being protected by law, could definitely be labeled as sensitive from the company's perspective. Data at endpoints is usually labeled as data in use. DLP systems responsible for data in use monitoring scan endpoints for any sensitive data and their movement (e.g., transferring to a USB drive).

There are many DLP systems, both commercial and open source. Some are cloud-based and offer realtime sensitive data detection APIs for a stream of unknown data. Others integrate tightly with the workflow of the company. Users can directly label sensitive documents in office suites with automatic recommendations by the system. Solutions also differ in the way of detecting data of interest. A list of conventional approaches to the data recognition task follows. [32]

- **Rule-based Recognition**
  This includes regular expressions, keywords, and other pattern matching techniques. It is suitable for database records, files, and data blocks where records are of similar structure.

- **Database Fingerprinting**
  This method searches only for exact predefined matches, such as specific credit card numbers. It has a low false positive rate but is not of much use in general scenarios.

- **Exact File Matching**
  This technique involves building a database of files labeled as sensitive and then searching for hash values of these files. While easy to implement, its main drawback is that it only works for the identical file and so any modified version will not be detected.

- **Partial Document Matching**
  Similar to the previous technique, but can detect even modified files to

some degree. It involves using multiple hashes for different parts of the document. That way, for example, if a part of it is pasted into an email, it can be detected.

- **Conceptual/Lexicon**
  A combination of dictionaries, rules, and other analyses detect content that cannot be strictly defined, such as insider trading. It is likely to generate a lot of false positives and false negatives.

- **Statistical Analysis**
  This technique uses machine learning, Bayesian analysis, and other statistical methods to find content that resembles sensitive data. It is useful where similar, but not exact documents have to be detected, e.g., a repository of engineering plans. Natural language processing (NLP) can be used to take context into account regarding text-based data. It is suitable for detecting sensitive information in data like emails or contracts. For example, the word ”cold” could indicate a medical condition or refer to the temperature in the room. Some of these techniques are also used for spam detection. While previous categories are unable to cope with such situations, statistical analysis is not optimal for large amounts of structured data (like databases) because of performance constraints.

The application developed in this thesis is on the border of privacy by architecture, and technical point control focused on rule-based and conceptual/lexicon data recognition. It can be used to verify that no sensitive data has been added to the database after application deployment (after it was designed with privacy in mind) or as a starting point for data anonymization in an unknown database. Another use case would be to detect sensitive data where they should not be according to company policy. Rule-based recognition, conceptual/lexicon recognition, and database fingerprinting were selected as suitable concepts for detection based on the definitions of sensitive data in the previous chapter.

Chapter 3

Technology

3.1 Regular expressions

A regular expression (also regex or regexp) is a notation that defines sets of character strings. When a particular string falls into the defined set, we say that the regular expression matches the string. One way of describing a regular expression is a sequence of characters. There are many different standards and variations that use different characters, and the same characters can have a different meaning in them. The most notable are Perl Compatible Regular Expressions (PCRE) and the IEEE POSIX standard (which in itself has different sets of compliance), but many programming languages have a syntax of their own, although more often than not heavily influenced by any of the two. The syntax can vary even across different versions of the same software. More about the complex topic of regex capabilities on various platforms can be found at [12]. Figure 3.1 shows an example of a PCRE regular expression.

Today, the term regular expression is used even when it actually does not denote a regular language. This is because, over the years, regular expression engines were augmented with additional features that allow them to recognize languages that cannot be expressed by a classic regular expression. One of these features is *backreferences*. *Backreferences* allow engines to use the result of the previous capture in runtime (figure 3.2 shows an example). As these additional features can make recognition significantly slower [8], it is essential to check whether they are actually needed for the project and, if not, select an engine that does not support them.

All regular expressions (in the original meaning of that term) can also

Figure 3.1: PCRE regular expression that matches any text with hexadecimal number in it

```
(0x|0X)[A-Fa-f0-9]+
```

Figure 3.2: PCRE regular expression using *backreferences*. First capture group (a|b) matches either a or b and then the token \2 matches whatever the result was. This is then repeated with \1. In the end, either aaaa or bbbb is matched.

```
((a|b)\2)\1
```

Figure 3.3: Regular expression *a(bb)+a* represented as a non-deterministic finite automata (NFA) [8]

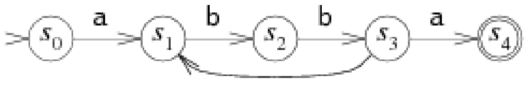
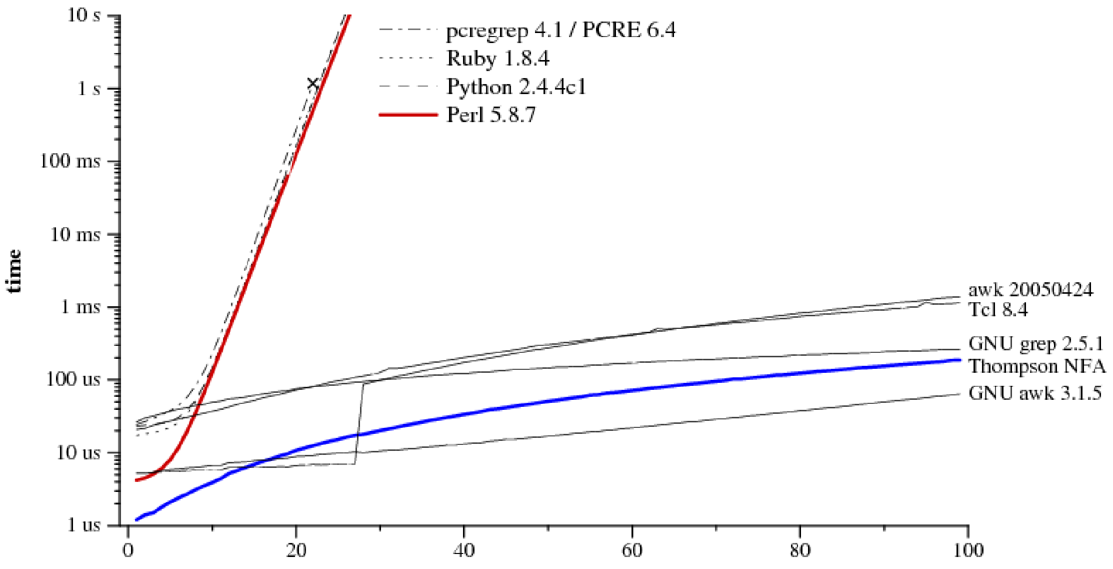


Figure 3.4: *"From the graph it is clear that Perl, PCRE, Python, and Ruby are all using recursive backtracking. The thick blue line is the C implementation of Thompson's algorithm, ..., Awk, Tcl, GNU grep, and GNU awk build DFAs."*[8]



be described as non-deterministic finite automata (NFA), and Thompson's construction algorithm can be used to convert any regular expression to NFA. This fact often is used when implementing a regular-expression engine. The part where a regular expression represented as a string is transformed into NFA or deterministic finite automata (DFA) is usually called compilation.

Regex engines can also be implemented using different methods and algorithms, one of which is backtracking. Backtracking is usually used to support those previously mentioned additional features. As was already said, for some regular expressions, it can be an order of magnitude slower [8]. These regular expressions are called pathological expressions, and on engines using such algorithms, they can require exponential time to process. Since there are no regular expressions that are pathological for NFA based engines, some argue that it is the best possible approach [8]. While implementation using Thompson's algorithm (NFA) requires $O(n^2)$ time, backtracking requires $O(2^n)$ time. [8] As figure 3.4 shows, that can present a significant gap in performance. Regular expression engines today usually use some combination of NFA, DFA, or backtracking together with custom optimizations. One should be aware of the implications for performance when selecting a regex engine for their project.

Now to address some of the commonly occurring configurations (or flags). Regex engines are, by default, *greedy.* This means that quantifiers like * match as many characters as possible. The opposite of *greedy* is *lazy* (or *ungreedy*), which matches the least number of characters possible. The behavior of the ˆ (start) and $ (end) tokens can usually be altered with a *multi-line* flag. When the *multi-line* flag is set, ˆ and $ also match the start and end of each line of the input. The default behavior is only to match the start and end of the input as a whole. A less intuitive flag is called *single-line* or *dotall.* When set, any instance of the dot token matches newline characters. Some engines also support stopping after the first match is found - *single-match*, the opposite of which is the *global* flag.

3.2 Choosing a Suitable Regular-expression Engine

Two main criteria were set for selecting the most suitable regex engine, with the first being raw performance. The second requirement is that the engine should support all patterns defined in chapter 1 and preferably no more. These two requirements are more similar to each other than they might appear since, as explained above, supporting more features leads to performance drawbacks, and if they are not needed, they should be avoided. When closely examining the patterns defined in chapter 1, it can be observed that there will not be a need for any additional features like backreferencing, other than classic regular expressions denoting a regular language. All patterns can, therefore, be processed into NFA or DFA, and the ideal regex engine will most likely use one of them in its implementation.

Benchmarking a regular expression engine is a difficult task. The reason is that modern engines can differ wildly in implementation details and optimizations. Therefore it is hard to come up with a combination of expressions that are not favoring any particular engine and test all of their aspects, which is the reason why the author decided not to do his own. The benchmark conducted in [4] has found the **Intel Hyperscan** engine as by far the best performing (figures 3.5 and 3.10). This benchmark was performed on a Ubuntu 16.04 operating system with Intel Core i5-5300U mobile CPU. It is important to note that the benchmark has received some criticism over the selection of regular

Figure 3.5: Benchmark of various regex engines (lower is better). Hyperscan is labeled as *hscan*. Note that Hyperscan is by far the fastest (approx. 3x less time than the 2nd). [4]

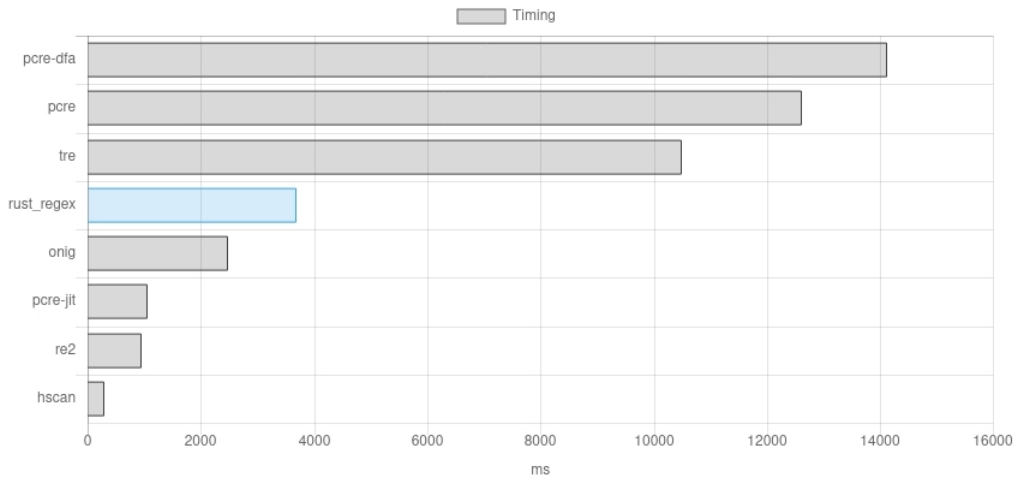
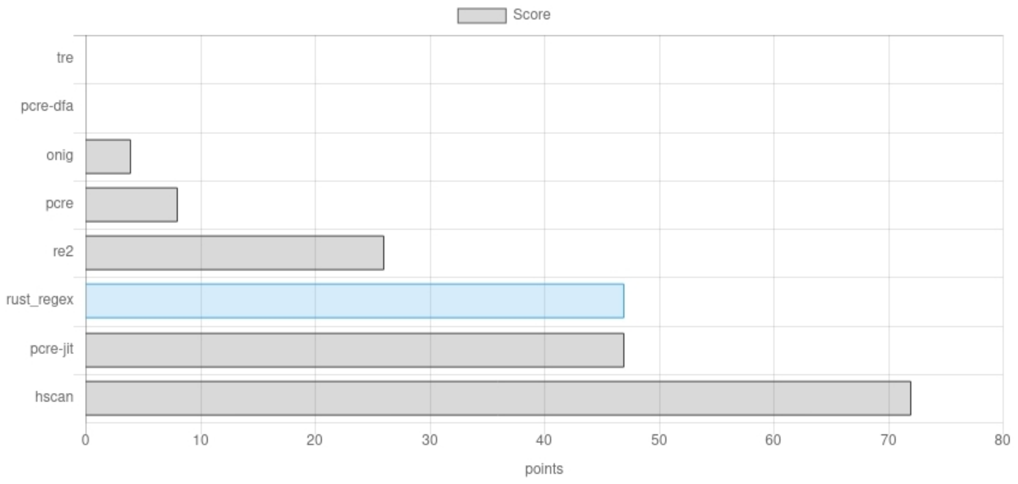


Figure 3.6: Benchmark of various regex engines (higher is better). Hyperscan is labeled as *hscan*. [4]



expressions and its conclusions, so the results should be taken with a grain of salt. Nonetheless, it is evident that the engine performs rather well.

While Hyperscan supports PCRE syntax, it does not support features like backreferences, and arbitrary lookaround asserts, which is good because, as said before, it is not needed. Internally, it is using both NFA and DFA as well as custom engines for special cases. It is also using single instruction, multiple data (SIMD) to handle multiple states in automata in parallel. This, however, has some implications for the system requirements. While Hyperscan's minimal requirements consist only of an x86 based CPU (32 or 64bit) with

Supplemental Streaming SIMD Extensions 3 (SSSE3) support, it can make use of some other instruction sets as well. These consist of Intel Streaming SIMD Extensions 4.2 (SSE4.2), the POPCNT instruction, Bit Manipulation Instructions (BMI, BMI2), and also Intel Advanced Vector Extensions 2 (Intel AVX2) [15]. While these instructions have their AMD counterparts, it is unclear whether the engine works correctly with them or if at all. To this date, there are no known benchmarks of this engine on an AMD CPU, and even its contributors state that Hyperscan works on AMD, but there could be performance degradation [17]. Since one of the goals of this thesis is that the application should be cross-platform, this question cannot be left unanswered.

Figure 3.7: Benchmark output on one of the tested systems.

```
C# Regex benchmark starting...
Expressions compiled in 39.17 milliseconds
[EMAIL] Iteration   0 took 3335.15 milliseconds
              <output omitted>
[EMAIL] Iteration 499 took 3062.56 milliseconds
[EMAIL] Average milliseconds for each iteration: 3550.65
[IPv4] Iteration   1 took 231.70 milliseconds
              <output omitted>
[IPv4] Iteration 499 took 240.83 milliseconds
[IPv4] Average milliseconds for each iteration: 269.85

Hyperscan benchmark starting...
[EMAIL] Compiled in 13.74 milliseconds
[EMAIL] Iteration   0 took 79.82 milliseconds
              <output omitted>
[EMAIL] Iteration 499 took 74.59 milliseconds
[EMAIL] Average milliseconds for each iteration: 66.10
[IPv4] Compiled in 10.33 milliseconds
[IPv4] Iteration   0 took 20.32 milliseconds
...
```

A simple application was programmed in .NET Core (C#) to provide a comparison between its native regular expression engine and Intel Hyperscan. Note that the purpose is not to compare these engines themselves but rather the difference in performance for Hyperscan on different CPUs where the performance of the native engine is used as a base point. Two simple regular expressions were constructed - IPv4 address and RFC 2822 email address, and the input text file was obtained from [18]. Efforts were made to make conditions for both engines as similar as possible. For C#, both expressions were compiled (since Hyperscan does it), and the multi-line flag was set. In Hyperscan, an additional left-most flag had to be set, because, by default,

Figure 3.8: Benchmark results for all tested CPUs in milliseconds (lower is better)

	.NET Core Regex (ms)		Hyperscan (ms)		
CPU	**Email**	**IPv4**	**Email**	**IPv4**	**Combined**
Intel i5-8300H	1,495.17	119.06	34.62	7.56	35.28
Intel Xeon E5-2630 v4	3,550.65	269.85	66.11	15.43	68.25
AMD Ryzen 3 3200G	1,870.14	139.06	36.58	10.29	38.97
AMD Ryzen 5 1600	2,128.40	161.26	39.99	10.08	42.51

it does not track the position of the matches as closely as C# engine does. Encoding on both platforms was set to UTF-8. Hyperscan was compiled from source to a shared object (.so) and loaded dynamically using Interop Marshaling.

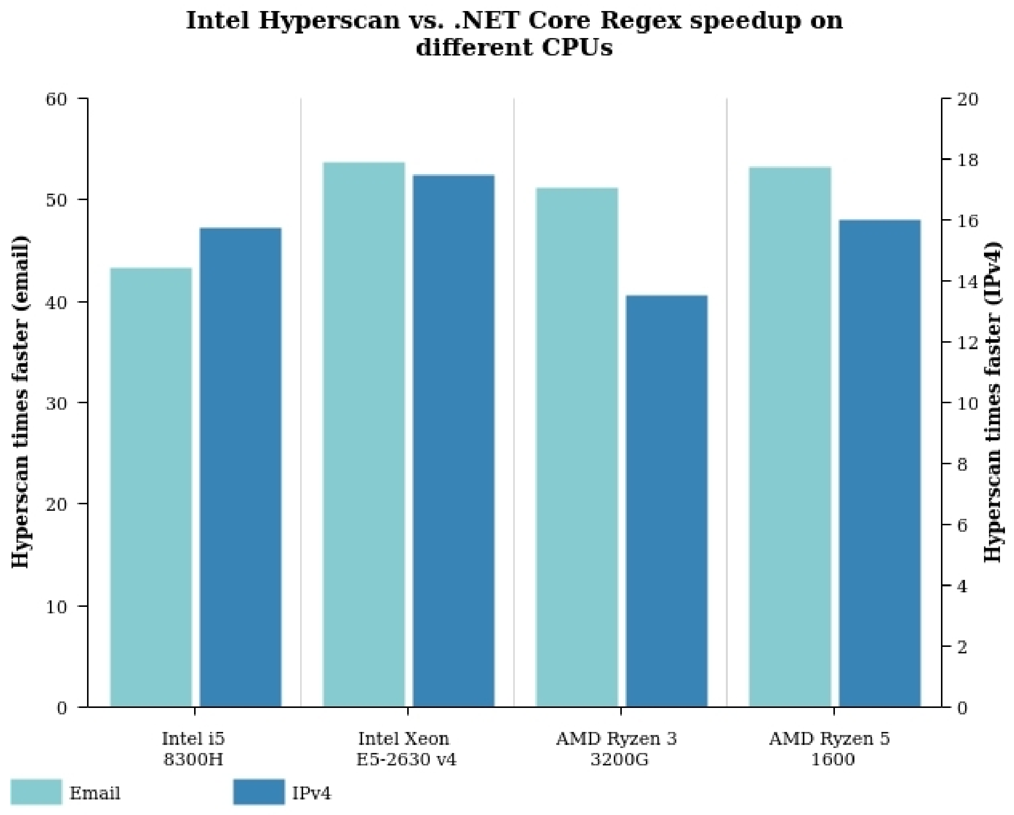
Four different CPUs were selected for benchmarking, of which two were from Intel and two from AMD. The recognition ran 500 times for each expression and engine. Times for each iteration were then averaged into a single value. The output for one of these runs is shown in figure 3.7. Table 3.8 shows complete results for each CPU, and figure 3.9 shows the relative performance gap between Hyperscan and the native engine on different CPUs. As the graph shows, on Ryzen 5, the performance gap was more significant than on the 8300H (Hyperscan was relatively faster on AMD compared to .NET Core regex in that very instance). Based on this simple benchmark, it can be concluded, that the Intel Hyperscan engine works reasonably well even with AMD CPUs. Also, as anticipated, Hyperscan was always significantly faster.

3.3 Databases

A database is an organized collection of structured data. Nowadays, databases are usually accompanied and tightly integrated with a database management system (DBMS). DBMS serves as a layer between the database and its users. A database with a DBMS is referred to as a database system, or in short, a database.

Databases can be divided into several categories based on how they represent the data in them. Probably the most notable would be the **relational database**. It is based on collections of data with pre-defined relationships between them. This data is usually modeled in rows and columns in a series of tables. Each column can store only a single type of data, and each row in a given table must have the same columns. Users can communicate with the database through queries. Most of the relational database management

Figure 3.9: Benchmark results show how many times faster Hyperscan is compared to the native .NET Core regex engine on each CPU.



systems support the **Structured Query Language (SQL)**, which is divided into four categories. Data query language (DQL) can retrieve specific data and cross-reference tables when doing so. Data definition language (DDL) is used to define new structures in a database like tables, columns, views (stored query), or indexes. Data manipulation language (DML) allows the user to add (insert), modify (update), or delete the data. Finally, data control language (DCL) is used to manage access control to a database. While there exists a standard for SQL, it is hardly ever implemented precisely as standardized, and although similar, SQL support differs across database management engines.

Closely related to relational databases is the concept of **ACID**, an acronym for atomicity, consistency, isolation, and durability. Atomicity means that each transaction is indivisible. Is either fails or succeeds as a whole as all changes as reverted in case of failure. Consistency ensures that the database must always be in a valid state after it is done. Valid state means that any constrains and relationships between the data are complied with. While transactions can be executed in parallel, isolation guarantees that the final state of the database after all transactions are executed is the same as if they would have been executed sequentially. Durability requires a transaction, that has once been committed, to remain committed even in case of a sudden failure. The vast majority of relational databases comply with ACID.

The opposite of ACID is **eventual consistency** or **BASE**, the shorthand for Basically Available, Soft state, Eventual consistency. Basically available means that while read and write operations to a database are available, there's no guarantee of consistency (reads might not get the latest write and writes might not persist). Soft state means that even when there are no write operations currently being executed, the state of the database might still be subject to change. Eventual consistency, as its name indicates, denotes that at some point after the last write operation, the database will eventually be in a consistent state.

While a BASE cannot offer the same reliability and consistency as ACID, it allows a database to perform better. When an organization must work with large amounts of unstructured data, a NoSQL database might be worth considering (most of the NoSQL databases do not comply with ACID, but there are exceptions). **NoSQL** databases model data in other structures than tables and columns. Databases that represent data as objects as in object-oriented programming are called **object-oriented databases**. These databases usually support some form of a query language. Attempts were made to standardize these languages as the Object Query Language (OQL). Another type of NoSQL database is a **graph database**. In it, the data is represented in graph structures with nodes and edges where nodes represent entities and edges relationships between them. Graph databases can also support a query language similar to SQL. **Document store** or **document-oriented database** system is a NoSQL database in which records have no uniform structure (records can have different "columns"), and types of records of individual columns can be different for each individual record. Records can also include records themselves (nested structure). These databases usually store and process data in formats like XML and JSON. **Key-value databases** or **key-value stores** are databases that can only store pairs of keys and values. They are not suitable for more complex data storage needs since they cannot model any relationships, but they can offer excellent performance for simple, more narrow use cases like embedded devices. **Wide column stores** or **extensible stores** are similar to key-value databases, with the exception that one key can have multiple values. These values are called columns, although they are not columns in the sense that relational databases use it. For each value, there can be a different number of columns with different names using different data types. Wide column stores can be seen as two-dimensional key-value databases. The last covered NoSQL database type in this list is a **search engine database**. Databases for search engines use a specialized, non-standardized structure that allows them to optimize search queries. In production, they can be accompanied by a more conventional database, where the data is stored permanently and then piped to a search engine database for indexation.

A broader category of databases are databases targeted at storing **big data** (large quantities of data). Database engines labeled as fit for big data are usually able to distribute the load on many computational units (servers). Big data storage solutions can usually fall into one or more categories of databases defined in the previous paragraph (both relational and NoSQL).

Databases also differ in what kinds of data they can hold. These are referred to as supported **data types**. The vast majority of databases supports at least

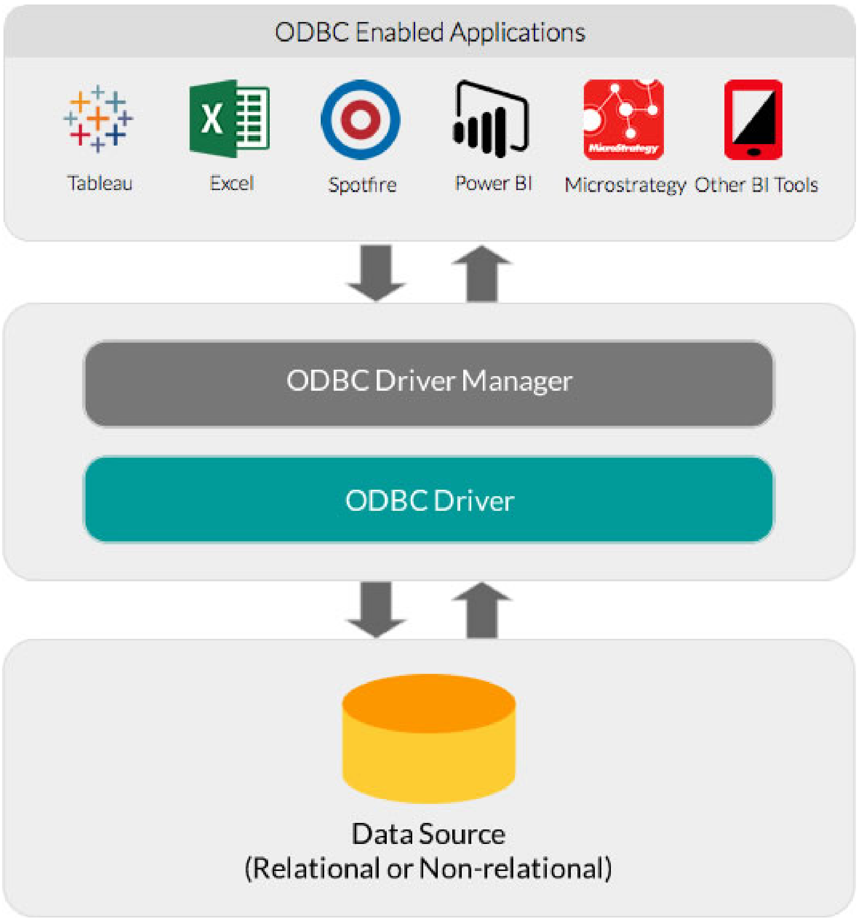
- integers,
- floating-point numbers,
- strings and single characters,
- types storing date and time,
- and binary data types.

Each mentioned category usually has multiple data types that differ in size and target use cases. Databases can support data types like location, images, or even let end users define their own.

Database management systems usually offer multiple ways of connecting to the database, both proprietary and standardized. **Open Database Connectivity (ODBC)** is a widely supported API standard for database connectivity. It is meant to expose most of the functionalities of a database management system. If ODBC is implemented in a database engine, we call the part responsible for translating API calls to native requests ODBC driver. The sole presence of an ODBC driver on a DMBS does not guarantee any functionality expected from a standard-compliant implementation. Different technologies have different capabilities, so many ODBC drivers do not implement all functionalities required by the standard or, conversely, implement more. While using the ODBC API for connecting to a database might be a preferred option (code reusability and simple implementation), one must consider other factors as well. The driver might not always be exposed, or its use restricted because of security concerns - in such cases, one must first enable it manually in DMBS. Also, compared to a native connection method, the performance of an ODBC driver might be worse because of the translation layer.

Many other standards exist with similar goals. Usually, they are more vendor-specific and do not have the same broad reach as ODBC does. Java Database Connectivity (**JDBC**) was developed on the basis of the ODBC, mainly targeted for use with the Java programming language (as opposed to ODBC, which is multi-language). JDBC can be used to connect to a database with ODBC support over JDBC-to-ODBC bridges and vice-versa

Figure 3.10: Open Database Connectivity (ODBC) architecture [39]



through ODBC-to-JDBC bridges. However, one should expect performance drawbacks when using such bridges. Object Linking and Embedding Database (**OLE DB**) is an API standard designed by Microsoft that allows accessing data from various data sources. Apart from products developed by Microsoft, it was also supported by Oracle and many others. OLE DB was announced as deprecated by Microsoft in 2011 and then undeprecated in 2017 since it remained popular despite its deprecation.

3.3.1 Target Database Management Systems

Several database management systems were selected for examination. PostgreSQL, MySQL/MariaDB, and Oracle DBMS are included because they are specified in the assignment. The selection of the rest was based on the usage and popularity from [9]. All of the engines bellow provide ODBC connectivity.

- **MySQL**

MySQL is an open-source relational database. It also supports document store as its secondary data model. Regular expressions are supported “similar to those used by Unix utilities such as vi, grep, and sed.” [20] This means that all patterns specified as regexes can be evaluated directly by the database engine which increases performance. In 2010, when acquired by Sun (later Oracle Corporation), it was forked by its original author to create the MariaDB project. While there are some differences between the two, they are not relevant from the point of sensitive data detection. For the sake of simplicity, MariaDB is not discussed any further in this thesis.

- **PostgreSQL (Postgres)**
  PostgreSQL is also an open-source relational database with a document store as its secondary data model. It is owned and developed by its community. It supports POSIX regular expressions with some features added on top. [26] Regular expressions with fewer features are also supported when using *LIKE* and *SIMILAR TO* operators.

- **Oracle DMBS**
  Oracle Database is a proprietary database management system owned and developed by the Oracle company. While its primary data model is relational, it also supports document store and graph store. Regular expressions support is compliant with the POSIX Extended Regular Expression. [21] Columns that contain sensitive data can be labeled as such together with a degree of confidentiality.

- **MSSQL Server**
  Microsoft SQL Server, like Oracle DMBS, is a proprietary database with its primary model being relational and its secondary models being document store and graph store. Regex support is, however, rather limited [23]. Using the *LIKE* keyword, one can use only a small subset of features of the likes of POSIX regular expressions. The result is that even something as simple as an IPv4 address cannot be specified appropriately. Support for advanced regular expression can be obtained, but only through a custom CLR function. Since specifying a CLR function essentially means writing to a database, it is deemed unsuitable for auditing. As well as Oracle DMBS, MSSQL also supports labeling columns as sensitive.

- **MongoDB**
  MongoDB is an open-source NoSQL database owned and developed by the MongoDB company, with document store being its primary data model. It supports PCRE regular expressions [19] and has some features of a search engine database.

- **Hive**
  Apache Hive is an open-source data warehouse focused on big data built on top of the Hadoop framework. Its only data model is relational. HiveQL, its query language, is an SQL-like language with regex support through its *RLIKE* operator. *RLIKE* has the full support of Java regular expressions. [31]

Companies nowadays offer databases in the cloud. Usually, these are either the same or slightly modified (optimized for distributed workloads) versions of "classic" database management systems. For example, Azure SQL is a cloud counterpart of MSSQL Server. From the standpoint of database crawling and sensitive data detection, these differences are not essential and are not addressed further in this thesis.

3.4 Database Crawling

To crawl (search) a database, one must first know its structure. For relational databases (or their relational parts), this is simple since database schema (structure) is known at all moments. Problems might arise when discovering the structure of a NoSQL database. Since the data does not have to be structuralized, even the database management system might not have the schema at its disposal. In such cases, the only option is to acquire the necessary amount of data to construct the schema on one's own. Performance must be taken into consideration, as any unnecessary operation could prolong the discovery beyond a reasonable timespan. In the following lines, previously selected databases are analyzed from the standpoint of schema discovery. Note that only original data sources like tables and columns are of interest. Data derived from them like views or virtual columns are not considered since they are necessarily based on the original data. Also, any information regarding relationships between entities is ignored, because the relationships themselves cannot be the storage of the sensitive data either.

MySQL
Since MySQL is a relational database, schema discovery is rather easy. In database metadata, table *INFORMATION_SCHEMA.TABLES* holds a list of all tables for a particular database, and *INFORMATION_SCHEMA.COLUMNS*, as would one expect, holds all columns. Figure 3.11 shows an SQL query for obtaining a list of all tables and then a list of all columns for one particular table.

PostgreSQL (Postgres)
Postgres makes information about tables and columns available in system

Figure 3.11: Query that obtains a list of all tables for database "db" and lists all columns for the table named "table" in MySQL

```
#Get all tables
SELECT table_name
FROM INFORMATION_SCHEMA.TABLES
WHERE table_schema = 'db';

#Get all columns for table 'table'
SELECT COLUMN_NAME
FROM INFORMATION_SCHEMA.COLUMNS
WHERE TABLE_SCHEMA = 'db' AND TABLE_NAME = 'table';
```

catalogs, in particular *pg_catalog.pg_tables*, *information_schema.tables* and *information_schema.columns*. Figure 3.12 demonstrates a practical example in SQL.

Figure 3.12: Query that obtains a list of all tables for database "db" and lists all columns for the table named "table" in PostgreSQL

```
#Get all tables
SELECT tablename from pg_catalog.pg_tables
WHERE schemaname = 'db';

#Get all columns for table 'table'
SELECT information_schema.columns
WHERE table_schema = 'db' AND table_name = 'table';
```

Oracle DMBS

When listing all tables in Oracle DMBS, one must ensure that the necessary privileges are obtained. The example query shown in figure 3.13 assumes that the user can access the *dba_tables* and *ALL_TAB_COLUMNS* tables. Note that virtual columns are ignored when *ALL_TAB_COLS* is used. They are only shown if *ALL_TAB_COLUMNS* is used.

MSSQL Server

The list of tables and columns is stored in
<database_name>.INFORMATION_SCHEMA.TABLE and
<database_name>.INFORMATION_SCHEMA.COLUMNS tables. The table type has to be filtered so that only *BASE TABLE* tables are selected. This ensures that views are ignored. Figure 3.14 shows an example query.

Figure 3.13: Query that obtains a list of all tables and lists all non-virtual columns for the table named “table” in Oracle DBMS

```
#Get all tables
SELECT table_name FROM dba_tables;

#Get all columns for table 'table'
SELECT COLUMN_NAME
FROM ALL_TAB_COLUMNS
WHERE TABLE_NAME='table';
```

Figure 3.14: Query that obtains a list of all tables and lists all columns for the table named “table” in MSSQL. *BASE_TABLE* ensures, that no views are selected.

```
#Get all tables
SELECT TABLE_NAME
FROM 'db'.INFORMATION_SCHEMA.TABLES
WHERE TABLE_TYPE = 'BASE␣TABLE'

#Get all columns for table 'table'
SELECT COLUMN_NAME
FROM 'db'.INFORMATION_SCHEMA.COLUMNS
WHERE TABLE_NAME = 'table'
```

MongoDB
MongoDB is a typical example of a NoSQL database with document store model, so as expected, the database structure is not as easily available. Using the *listCollections* command, one can obtain collections (analogy of tables). In collections, MongoDB stores documents, and each of these documents can have its own fields (analogy of columns). Fields can have a nested structure (similar to JSON objects) and also a different data type for each document. Getting a complete schema (including nested structures) is a non-trivial task, so much so that open-source projects emerged with that goal in mind. [38] Figure 3.15 shows an example of discovering all fields for one collection without the support for nested types in the Mongo shell.

Hive
Hive is built on relational principles (although it is not a relational database), so obtaining a schema is not as hard as for typical NoSQL database. Since version 3.0.0, INFORMATION_SCHEMA is supported as long as one has access to its metadata database (metadata are stored in a separate relational

Figure 3.15: Code snippet in Mongo shell that gets a list of all fields on the first level of depth for the collection named “collection” [11]

```
db.collection.aggregate([
  { "$project": {
    "data": { "$objectToArray": "$$ROOT" }
  }},
  { "$project": { "data": "$data.k" }},
  { "$unwind": "$data" },
  { "$group": {
    "_id": null,
    "keys": { "$addToSet": "$data" }
  }}
])
```

database). [16] In that case, requesting a list of tables and columns would be similar to MySQL (shown in figure 3.14). If the metadata database cannot be accessed, the Beeline shell can be used instead (figure 3.16).

Figure 3.16: Listing all tables in database “db” and then all columns for table “table” in Beeline shell (Apache Hive)

```
#Get all tables
use db;
show tables;

#Get all columns for table 'table'
show columns in 'table';
```

3.5 C# programming language and .NET Core Framework

C# is a cross-platform type-safe and object-oriented programming language, currently owned and developed by Microsoft. It is commonly used with .NET Framework, which runs primarily on the Microsoft Windows operating system. Its free, open-source, and cross-platform counterpart, **.NET Core**, runs on macOS, GNU/Linux, and MS Windows.

While .NET Core supports GUI frameworks like Windows Presentation Foundation (WPF) and Universal Windows Platform (UWP), these frameworks themselves are not cross-platform. Therefore to create a cross-platform application with GUI, one must resort to third-party frameworks and libraries.

Well established frameworks like **Qt** or **GTK** are stable and offer high performance, but other problems arise. In the case of Qt, it is problematic (expensive) licensing for commercial projects. GTK, on the other hand, is entirely free but looks alien (at least by default) on other platforms than GNU/Linux. Platforms like Electron are simple to integrate and easily portable, but very resource-heavy since they need an entire web browser to run. **Avalonia** framework is free for commercial use, cross-platform, provides easy to use XAML dialect similar to WPF and the .NET Foundation (founded by Microsoft) supports its development. However, at the time of writing, it still has not reached the production-ready stage, mainly due to insufficient documentation. For this reason, GTK was selected as the most optimal UI framework to achieve the goals of this thesis.

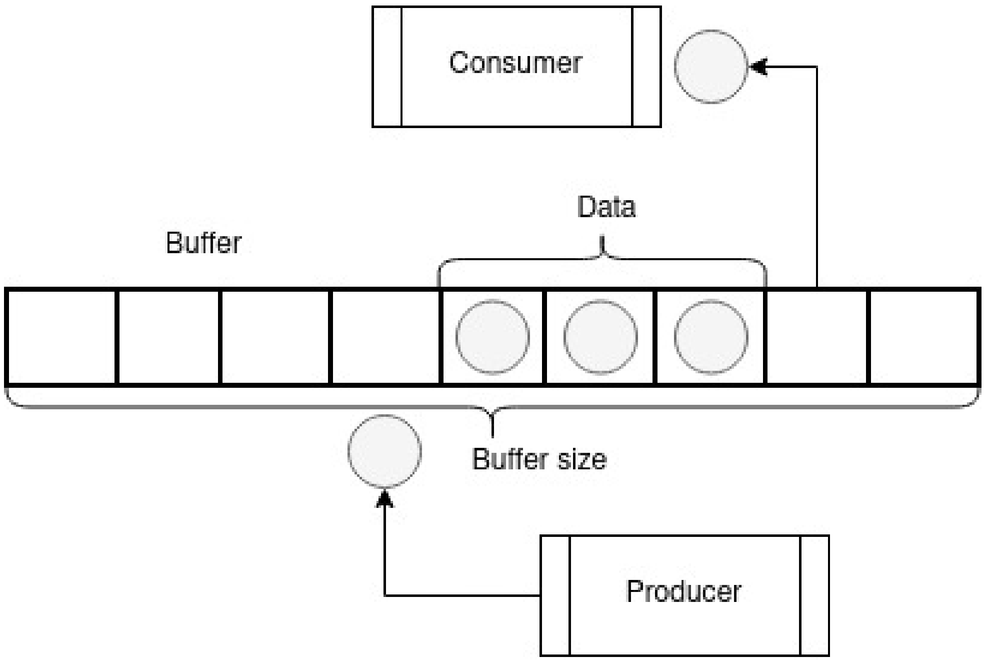
.NET Core supports ODBC through a simple to use *OdbcConnection* class, and MSSQL Server can be accessed natively using the optimized *SqlConnection* class. Packages providing additional features and capabilities can be obtained through the NuGet package manager, which also manages updates. All target databases from the list above can be connected to from .NET Core application with either a package from NuGet, a connector from a vendor's website, or a generic ODBC connection. NuGet offers packages MySql.Data for Mysql, Npgsql for PostgreSQL, ODP.NET for Oracle DMBS, to name but a few.

If there is a need to call native code from a dynamic-link library (or a shared object), .NET Core (C#) has that covered as well through *DllImportAttribute*. Interop Marshaling governs how data is transferred between managed and unmanaged memory during calls. While relatively simple for built-in data types like integers and booleans, problems might arise with custom classes and pointers. Instead of resorting to unsafe code, which potentially introduces security risks, one should rather create proper wrappers around imported functions and objects and let the language runtime handle the memory. Since doing this correctly might be tedious and time-consuming, projects like [25] were created to help with the process.

3.6 Producer-consumer Pattern

Producer-consumer is a synchronization problem, where several threads exchange data through a shared memory with limited speed and capacity (a buffer). These threads can be divided into two groups, called producers and consumers. Producers fill the buffer with equally sized blocks of data until there are no more data blocks to produce. If the buffer reaches its capacity, producers wait. Consumers remove data blocks from the buffer to process them, and, by analogy, if there's no data in it, they wait until there is. In the end, producers notify consumers that no more data will be produced. There may be additional requirements on the order in which data blocks are being produced or consumed.

Figure 3.17: Producer-consumer pattern (one producer, one consumer)



The producer-consumer pattern is used in many programs, one of which is the piping mechanism in Unix based operating systems. When two programs are divided by a pipe, one produces data, and the other consumes them, for example, *"cat file.txt | grep text"* in GNU Bash. Program *cat* reads file.txt and fills the buffer with text and program *grep* processes this data the moment it is available.

There are many ways in which the pattern can be implemented. Incorrect implementations can suffer from race conditions, deadlocks, or introduce bottlenecks due to improper usage of threading capabilities on the given system (busy waiting). In .NET and .NET Core, one can utilize the TPL Dataflow Library by Microsoft. It works on all target platforms specified in the goals of this thesis and takes care of many problems internally. Using *Dataflow.ITargetBlock<TInput>* and *ISourceBlock<TOutput>* objects, one can easily share data between producer and consumer methods. After the producer is finished, it calls the *Complete* method to let the consumer method know that no more data will be produced. Figure 3.18 shows a simple template for one producer and one consumer exchanging bytes.

Figure 3.18: A template for a simple producer-consumer pattern in C# .NET [1]

```
//Produce data
static void Produce(ITargetBlock<byte[]> target)
{
    while (...)
    {
        ...
        target.Post(buffer);
        ...
    }
    target.Complete();
}

//Consume data
static async Task<int> ConsumeAsync(ISourceBlock<byte[]> source)
{
    while (await source.OutputAvailableAsync())
    {
        ...
        byte[] data = source.Receive();
        ...
    }
    return bytesProcessed;
}

static void Main(string[] args)
{
    var buffer = new BufferBlock<byte[]>();
    var consumer = ConsumeAsync(buffer); //start consuming
    Produce(buffer); //produce data
    consumer.Wait();
}
```

Chapter 4

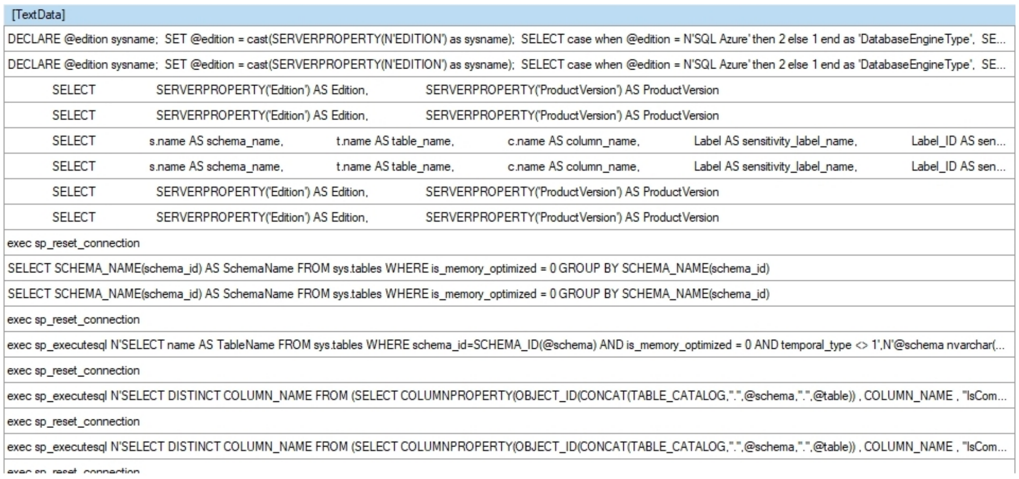
Commercial Solutions

A few commercial solutions available for searching sensitive data in a database were selected for examination, mainly for confirmation, that the algorithm designed in this thesis is the best possible approach performance-wise. As was already mentioned, compared to other database engines, Microsoft SQL (MSSQL) Server has considerably poorer regular expression support. There is a way of enabling better regex support (user-defined CLR function), but it involves writing to the database which, in any auditing, should be avoided.

As one would expect, selecting all data and evaluating them locally is notably slower than using the power of a database engine to do the evaluation. And since with MSSQL, it is impossible to use regex to match something as simple as an IP address, it will be used as a primary platform for evaluation. If there's a way to avoid evaluating all data locally, it is reasonable to expect that professional solutions will surely do so, as it would be a lot faster. Products to be tested were selected based on popularity (highest ranking in the search engines) and also in consideration of how hard it was to obtain a free trial version to test its capabilities. In a few cases, the sales representatives of the respective companies required the author of the thesis to sign an NDA before providing the software. Since this would prevent him from reverse-engineering it or publishing any results from research, these requests were politely declined.

The target database for all three test subjects has been chosen to be the AdventureWorks2017 database [5]. This is a sample published by Microsoft to show how to design a database. Since it contains a lot of sensitive data like names, birthdates, and email addresses, any tools that claim the ability to scan a database for such information should be able to detect it. To ensure that the tool scans the content and not just column names, two versions of the database are to be used. The original is to be left unchanged, and the second one is to have its columns renamed to arbitrary names by the author of this thesis. One can easily capture queries sent to the MSSQL server using the SQL Server Management Studio (SSMS) XEvent Profiler. These captured queries are then to be used to determine what data left the server.

Figure 4.1: Queries sent by the Data Discovery & Classification tool - SQL Server Management Studio XEvent Profiler (cutout)



```
[TextData]
DECLARE @edition sysname;  SET @edition = cast(SERVERPROPERTY(N'EDITION') as sysname);  SELECT case when @edition = N'SQL Azure' then 2 else 1 end as 'DatabaseEngineType',  SE...
DECLARE @edition sysname;  SET @edition = cast(SERVERPROPERTY(N'EDITION') as sysname);  SELECT case when @edition = N'SQL Azure' then 2 else 1 end as 'DatabaseEngineType',  SE...
        SELECT          SERVERPROPERTY('Edition') AS Edition,           SERVERPROPERTY('ProductVersion') AS ProductVersion
        SELECT          SERVERPROPERTY('Edition') AS Edition,           SERVERPROPERTY('ProductVersion') AS ProductVersion
        SELECT          s.name AS schema_name,          t.name AS table_name,           c.name AS column_name,          Label AS sensitivity_label_name,            Label_ID AS sen...
        SELECT          s.name AS schema_name,          t.name AS table_name,           c.name AS column_name,          Label AS sensitivity_label_name,            Label_ID AS sen...
        SELECT          SERVERPROPERTY('Edition') AS Edition,           SERVERPROPERTY('ProductVersion') AS ProductVersion
        SELECT          SERVERPROPERTY('Edition') AS Edition,           SERVERPROPERTY('ProductVersion') AS ProductVersion
exec sp_reset_connection
SELECT SCHEMA_NAME(schema_id) AS SchemaName FROM sys.tables WHERE is_memory_optimized = 0 GROUP BY SCHEMA_NAME(schema_id)
SELECT SCHEMA_NAME(schema_id) AS SchemaName FROM sys.tables WHERE is_memory_optimized = 0 GROUP BY SCHEMA_NAME(schema_id)
exec sp_reset_connection
exec sp_executesql N'SELECT name AS TableName FROM sys.tables WHERE schema_id=SCHEMA_ID(@schema) AND is_memory_optimized = 0 AND temporal_type <> 1',N'@schema nvarchar(...
exec sp_reset_connection
exec sp_executesql N'SELECT DISTINCT COLUMN_NAME FROM (SELECT COLUMNPROPERTY(OBJECT_ID(CONCAT(TABLE_CATALOG,''.'',@schema,''.'',@table)) , COLUMN_NAME , ''IsCom...
exec sp_reset_connection
exec sp_executesql N'SELECT DISTINCT COLUMN_NAME FROM (SELECT COLUMNPROPERTY(OBJECT_ID(CONCAT(TABLE_CATALOG,''.'',@schema,''.'',@table)) , COLUMN_NAME , ''IsCom...
exec sp_reset_connection
```

The first solution to test is the **Data Discovery & Classification** tool from Microsoft for MSSQL Server. It is built-in since the 2012 version and available for free in the SQL Server Management Studio (SSMS). The description on the product page states that *"the classification engine scans your database and identifies columns containing potentially sensitive data. It then provides you an easy way to review and apply the appropriate classification recommendations, as well as to manually classify columns"* [36]. That sounds promising; based solely on that description, it could be a solution for the task of this thesis on its own. Let us verify these claims.

For the database with original column names, the tool reported 56 columns as recommended for classification. This meant that it indeed detected these columns as potential carriers of sensitive data. Inspection of the captured SQL queries showed that except for some metadata, it received table and column names onlu; not any content of the columns themselves (figure 4.1). For verification, the second classification session was launched on the database with renamed columns. As expected from the capture, the tool failed to detect any sensitive data at all. It is, therefore, safe to state, that the Data Discovery & Classification tool is not suitable for sensitive data detection in an unknown database on its own, and there are no valuable crawling algorithms or techniques to be discovered observing it.

Much the same could be said about the second examined solution, the **Idera SQL Column Search**, which is available as part of the Idera SQL Compliance Manager package [35]. Just as with the previous test case, it was tested on both original and renamed versions of the sample database, and its queries were captured. From the SQL queries alone, it is evident that the tool only searches column names (figure 4.2). Regular expressions

Figure 4.2: One of the queries captured when analyzing Idera SQL Column Search. Note expressions like %credit% or %card%

```sql
SELECT
db_name() as DatabaseName, SCHEMA_NAME(t.schema_id)
+'.'+t.name AS SchemaTable,(SUM(a.used_pages) * 8.0 )
/ 1024.0 AS UsedDataSpaceMB, COUNT(DISTINCT c.Name) AS
MatchedColumns, (SELECT SUM(row_count) FROM
sys.dm_db_partition_stats ps WHERE ps.object_id =
OBJECT_ID(SCHEMA_NAME(t.schema_id)+'.'+t.name ) AND
(index_id = 0 or index_id = 1)) AS RowsCount FROM
sys.tables AS t INNER JOIN sys.columns c ON t.OBJECT_ID
= c.OBJECT_ID INNER JOIN sys.indexes i ON t.OBJECT_ID
= i.object_id INNER JOIN sys.partitions p ON i.object_id
= p.OBJECT_ID AND i.index_id = p.index_id INNER JOIN
sys.allocation_units a ON p.partition_id = a.container_id
WHERE
(   c.name LIKE '%date%' COLLATE Latin1_General_CI_AS
or  c.name LIKE '%dob%' COLLATE Latin1_General_CI_AS
or  c.name LIKE '%dob%' COLLATE Latin1_General_CI_AS
or  c.name LIKE '%email%' COLLATE Latin1_General_CI_AS
or  c.name LIKE '%code%' COLLATE Latin1_General_CI_AS
or  c.name LIKE '%credit%' COLLATE Latin1_General_CI_AS
or  c.name LIKE '%card%' COLLATE Latin1_General_CI_AS
```

such as %credit%, %card%, or %email% used in the query mean, that any column with a name containing these words will be flagged. This fact has been confirmed by comparing columns detected when analyzing the renamed version of the database. Whereas for the original 143 columns were identified using the default settings, none were identified when analyzing the renamed one.

The last inspected product is the **Sensitive Data Discovery** Solution by DataSunrise [33], and it is arguably the most advanced of the three. Its web interface presents the user with a wide range of settings and preferences like security standards (e.g., GDPR or HIPAA) or which particular patterns to search for. It can also be specified as to how many rows to search for each column and the minimal amount of matches required for the column to be flagged (figure 3). The user is also able to add patterns of their own using regular expressions (figure 4) or lexicons like a list of world cities. As expected, the tool was able to detect sensitive data even in the database with renamed columns and tables. From the captured queries (figure 5), it can be observed that most of the data is pulled from the database server and evaluated locally. If the user selects a 100 rows to be searched from each column, the software

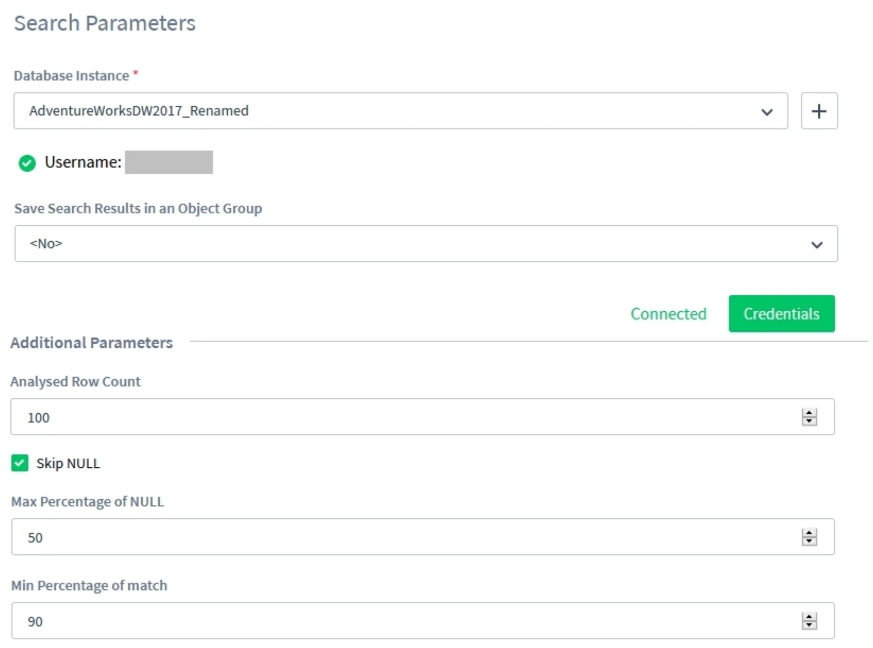
Figure 4.3: Some of the queries captured when analyzing Sensitive Data Discovery. C[number] and T[number] are the names of the renamed columns and tables.

```
...
declare @p1 int  set @p1=NULL  exec sp_prepare @p1
output,NULL, N'SELECT␣TOP␣100␣[C1],CAST([C2]␣as␣char)
AS␣[C2],[C3],[C4],[C5],[C7],[C8],[C9],[C10],[C11],[C12],
[C13],[C14],[C15],[C16],[C17],[C18],[C19],[C20]␣FROM
[DBO].[T6]',1 select @p1 exec
sp_execute 105
exec sp_unprepare 105 declare @p1
int set @p1=NULL exec sp_prepare @p1 output,NULL,
N'SELECT␣TOP␣100[C1],[C2],[C3]␣FROM␣[DBO].[T7]',1
select @p1
exec sp_execute 106
exec sp_unprepare 106
declare @p1 int  set @p1=NULL  exec sp_prepare @p1
output,NULL,N'SELECT␣TOP␣100␣[C1],[C2],[C3],[C4],[C5],
[C6],[C7],[C8],[C10],CAST([C11]␣as␣char)␣AS␣[C11],
CAST([C12]␣as␣char)␣AS␣[C12],[C13],[C14],[C15],[C16],
[C17],[C18],[C20],[C21],[C22],[C23],[C24],[C27],
CAST([C28]␣as␣char)␣nAS␣[C28],CAST([C29]␣as␣char)
AS␣[C29],[C30]␣FROM␣[DBO].[T8]',1
select @p1
...
```

request 100 top rows, rather than 100 random rows. Consider the wide variety of data stored in any database; this might not be the best choice as the top n rows may not be representative of the whole column. The aforementioned aside, the tool seems to be very capable and could fulfill at least some of the tasks set by this thesis.

In conclusion, it is evident that there most likely is no way to search an MSSQL database thoroughly without processing at least some data locally. It also seems that patterns specified as regular expressions represent the right balance between user-friendliness, extensibility, and performance. This simple analysis also served the author of this thesis as inspiration for class and UI design.

Figure 4.4: Setting up a new search using DataSunrise Sensitive Data Discovery



Search Parameters

Database Instance *

AdventureWorksDW2017_Renamed

✓ Username: [redacted]

Save Search Results in an Object Group

<No>

Additional Parameters

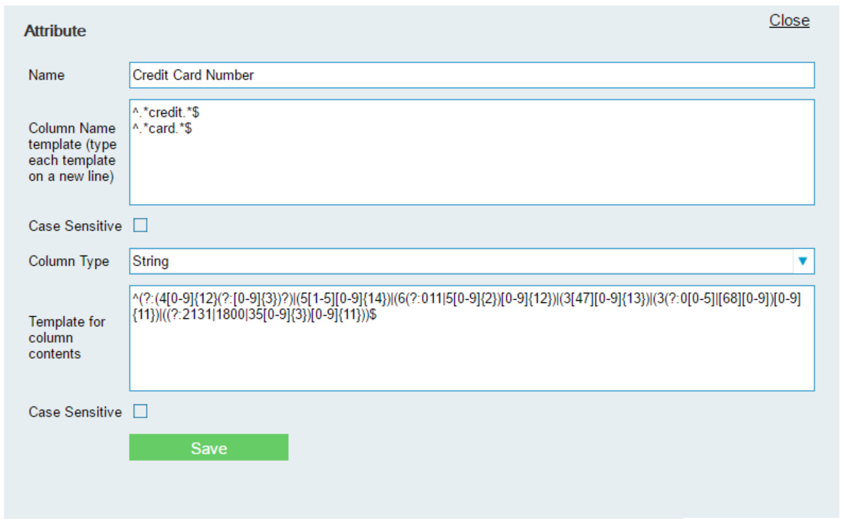
Analysed Row Count: 100

✓ Skip NULL

Max Percentage of NULL: 50

Min Percentage of match: 90

Figure 4.5: Adding new pattern in DataSunrise Sensitive Data Discovery [34]



Name: Credit Card Number

Column Name template (type each template on a new line):

```
^.*credit.*$
^.*card.*$
```

Case Sensitive ☐

Column Type: String

Template for column contents:

```
^(?:(4[0-9]{12}(?:[0-9]{3})?)|(5[1-5][0-9]{14})|(6(?:011|5[0-9]{2})[0-9]{12})|(3[47][0-9]{13})|(3(?:0[0-5]|[68][0-9])[0-9]
{11})|((?:2131|1800|35[0-9]{3})[0-9]{11}))$
```

Case Sensitive ☐

Chapter 5

Design

5.1 Design Objectives

- **Pattern Versatility**
  Users should be able to specify patterns in a universal and unified manner. If support for a new database is added, patterns that have already been specified should work on that new database without the need to edit them. E.g., if there is already a pattern for an IP address and it is working for MySQL server, after adding support for Oracle Database, this very pattern should work for it too. This also applies for all metadata that the pattern holds, like security standard or percentage of matches required.

- **Extensibility**
  Adding support for new databases should not require any of the code to be rewritten, only added. E.g., if the application does not support the SQLite database, a programmer should be able to add its support with the least possible effort, and all of the previously defined configurations, for example patterns, should work with it.

- **Computation Offloading**
  Routes all computationally intensive operations to the target database. E.g., instead of selecting all rows, and then evaluating them locally, construct a query that returns just the results.

5.2 Patterns and Crawling Algorithms

As shown previous, crawling a database mainly consists of discovering its data structure. When this structure is discovered, one can start searching the content itself. The first idea was to search entities considering the presence of other entities. In practice, this would mean that a location pattern could be defined as a rule, where two decimal columns with the same precision and range are in the same table. Alternatively, a table that contains a column with first names is arguably more likely to contain columns with last names, however this idea was dropped as location detection was the only meaningful example the author could manifest. After all, if there were a way of detecting a column with names in the first place, the surname column would likely be detected using that very method as well thus rendering the post-analysis useless. Also, some of the patterns defined might not be easily transferable to other database types begging the question how the location example would scale in case of a document store database where each entity may have different record types? Finally, this idea could also lead to exponential time complexity (comparing all columns to all columns), and complicated design with uncertain benefits.

The second approach is noticeably more straightforward: all entities should be searched separately with results from one having no impact on the other. This simplifies the architecture significantly and also allows defined patterns to be used on different database types unaltered (**pattern versatility**). Moreover, searching an entity for established patterns can usually be executed directly on a database using an appropriate query, fulfilling the **computational offloading** requirement. For a relational database, it essentially means searching each column's content for all applicable patterns.

A minimal amount (percentage) of records matched can be used to deal with false positives. For example, a column would be labeled as sensitive only when at least 10 percent of its rows contain first names. On the other hand, for a credit card number, this would be zero, since a credit card number is unique and less prone to false positives.

Based on the data categorized and described in chapter 1, several pattern classes have been defined. These were also inspired by the commercial solutions analyzed in chapter 4.

- **Regex Pattern (content)**
  A regular expression specifies this pattern. Any data with defined structures, such as phone numbers or email addresses, can be detected with a given regular expression. The pattern can hold multiple regular expressions, each specialized for different database management systems. In the following text, such patterns are referred to as **specialized patterns**. By the nature of regular expressions, this pattern is targeted on character-based data types. Still, it can also be used on other data types like numbers as numbers can be converted into character strings. As conversion to string is usually slow, this ability should be used with caution.

- **Structure Name**
  Since structure names can sometimes give more information than the

content itself (e.g., a column named birthdate), this pattern is used for this very case. As in the previous example, a regular expression is suitable for the detection of any possible structure names.

- **Lexicon**
  This pattern defines a list of possible values that the value may contain for it to be recognized as a match. Think of a list of all medical conditions to detect sensitive medical data or a list of cities to detect a physical address. Execution directly on a database should be preferred, but if it is not possible (or slower), the list can be converted to regular expression for local evaluation.

- **Integer Range**
  Sensitive data can also take the form of integers. While generally frowned upon, data like phone numbers can be stored as integers instead of strings, and the tool must be able to detect them. This pattern specifies the range in which the number must be in order to be recognized. For example, a valid Czech mobile phone number (without the national prefix) ranges from 600,000,000 to 609,000,000 or from 720,000,000 to 799,000,000. This pattern can be executed directly on a database and is always faster than conversion to string with regular expression evaluation.

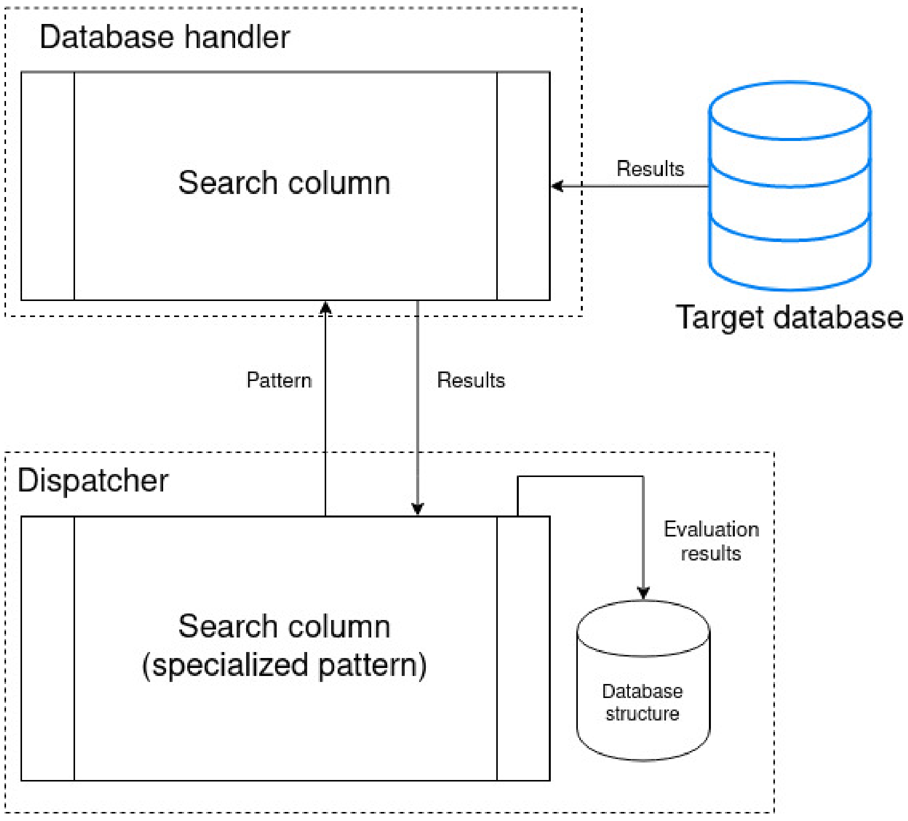
- **Decimal Range**
  Previously mentioned location could be detected using this pattern. It allows both value range specification (min and max value) and also the range for the decimal places. Location is usually specified as latitude (value from -90 to 90) and longitude (-180 to 180) with decimal places ranging from 5 to 9.

- **Date Range**
  This pattern can be used to detect dates that are generally expected to fall within defined range. Depending on a country's demographics, most adult birth dates fall between 1950 and 2000 (at the time of writing). This helps the pattern to distinguish from, for example, a date when an employment contract was signed.

It is important to note that these patterns have some disadvantages. One of them is the inability to work with checksums. For example, the International Mobile Equipment Identity (IMEI) number uses the Luhn algorithm to calculate its last verification digit. Because a 15 digit number is (arguably) unique and IMEI is frequently used without the last digit, the author of this thesis did not consider it essential.

To effectively manage a more extensive collection of these patterns, one needs metadata to label them and nested structures to group them. This is solved using **pattern groups**. One group can store multiple patterns of different kinds. For example, an email address group could store a structural pattern to detect column names containing the word “email” and also a corresponding regex pattern for the content. One level above is **information type**, which can store multiple groups (e.g., information type “Personal” stores pattern groups “email” and “phone number”). For each pattern, the **target language** can be specified. Multiple **compliance standards**, like “GDPR - sensitive,” can be assigned to each group.

Figure 5.1: Remote evaluation



5.3 Database Structure and Results Representation

After the database structure is discovered, it has to be stored. For that, a suitable representation that fits all target database management systems is required. The first idea was to have a structure that is general enough to contain both relational and NoSQL databases, and while it appeared feasible at first, it was dismissed after a few design attempts. The main issue with this approach is that these technologies are fundamentally different, with data being stored, defined, and accessed in incompatible ways. One output format would not work for both relational and NoSQL, as it would make things very confusing for the user. Even worse, one unified structure could have a negative impact on performance since it would make some optimizations hard or impossible to implement. Consider a MongoDB's document store nested structure that has to be "flattened" into single tables with all distinct fields enumerated as columns. While the end result resembles a relational database (and thus a unified structure is achieved), it would have been far better to process it in its native way without the overhead.

For these reasons, it was decided that while patterns can be used easily by any database type, **implementations of different database types should be separated.** Because of time constraints, **only relational database support is designed and implemented in this thesis.**

Replicating a relational database structure is rather simple. As it always consists of tables and columns, objects with the same names and hierarchy can be used to store them. Results can then be attached to each column since they are searched separately.

5.4 Search Dispatcher and Database Handlers

In order to satisfy the extensibility requirement, the design must separate the parts responsible for the connection to databases, as these are subject to change or extension. This is solved using the **database handlers** and the central **dispatcher**. Database handlers present an intermediate layer between a particular database management system and other parts of the application. They provide a unified interface for the dispatcher, a central layer that manages the search process and holds all information related to it. For the separation to work correctly, all database handlers must be indistinguishable from the perspective of the dispatcher. Figures 5.1 and 5.2 show the two main functions of the dispatcher.

The first function provides the handler with a pattern that is to be executed on the target database for the particular column. The handler then returns results, and these are processed, stored, and interpreted by the dispatcher (**remote evaluation**). The second function is the local pattern evaluation. A dispatcher requests the handler to start producing data (rows) for the given column into the shared buffer. This data is then processed simultaneously (and in parallel for each pattern) by the dispatcher using the Intel Hyperscan engine (**local evaluation**). This was described earlier as the producer-consumer pattern. The dispatcher is also responsible for searching the database structure names (evaluating structural patterns) as the structure is in its possession.

Since it is not uncommon for enterprise databases to contain billions of rows, searching all of it in a reasonable time-span is not a realistic expectation to have. The database handler is responsible for providing a statistically sound (representative) sample for the requested percentage of data. Results from searching this sample can then be used as if the entire column was reached.

Finally, figure 5.3 outlines the whole database scanning process in different stages.

Figure 5.2: Local evaluation

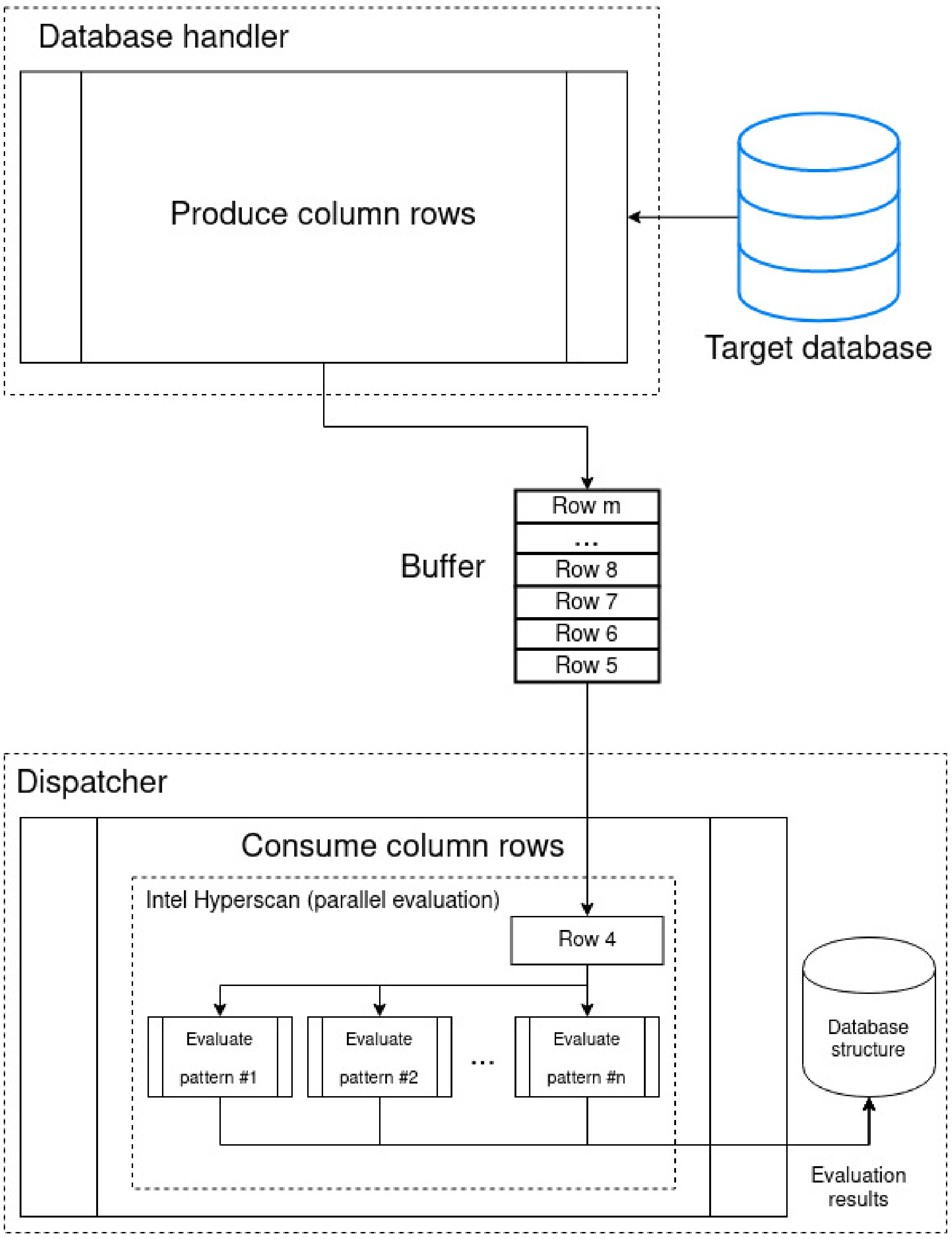
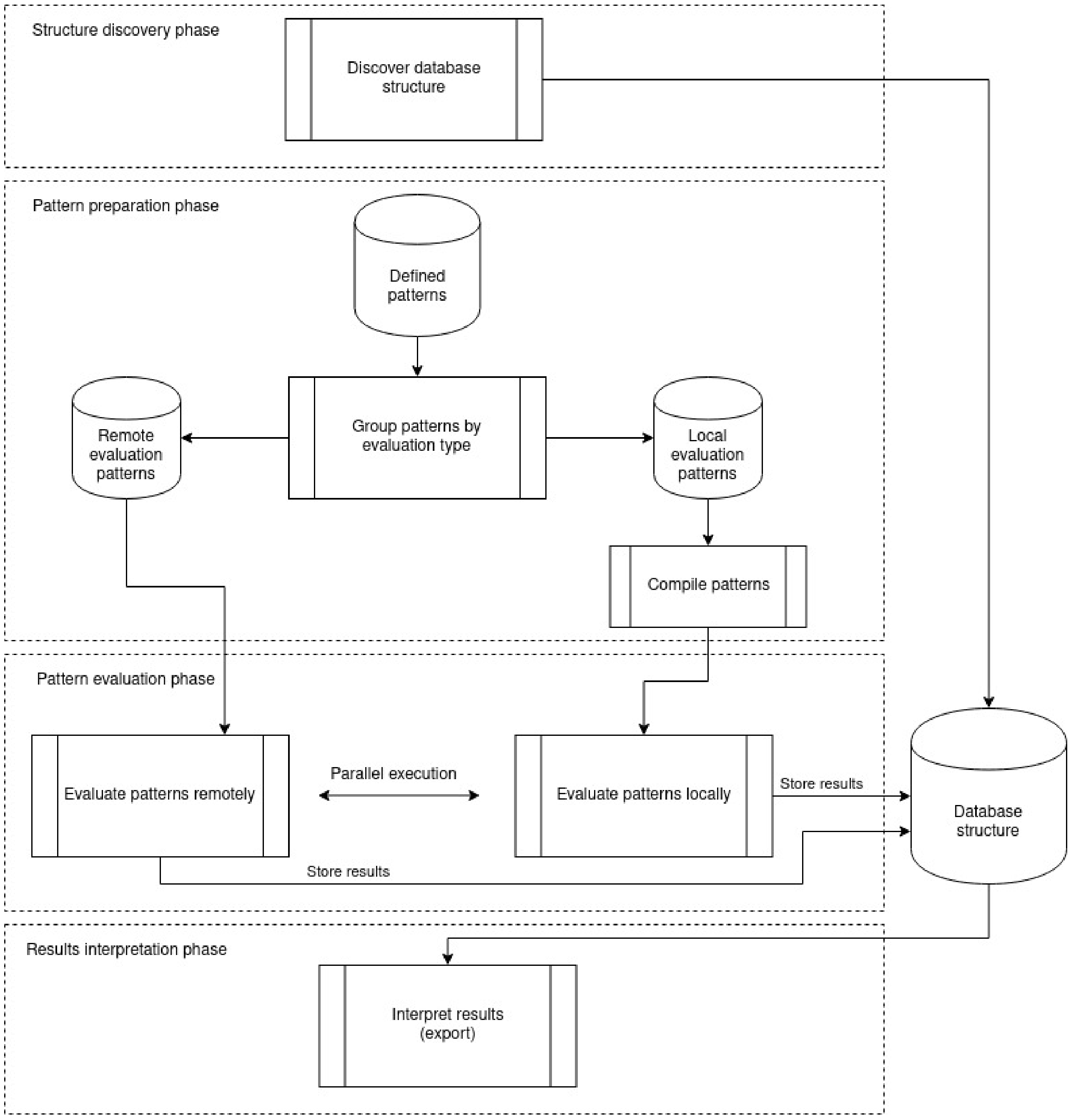


Figure 5.3: Scan process


Structure discovery phase
Discover database structure
Pattern preparation phase
Defined patterns
Remote evaluation patterns
Group patterns by evaluation type
Local evaluation patterns
Compile patterns
Pattern evaluation phase
Evaluate patterns remotely
Parallel execution
Evaluate patterns locally
Store results
Database structure
Store results
Results interpretation phase
Interpret results (export)

Chapter 6

Implementation

6.1 Patterns

The class diagram in figure 6.2 shows the implemented classes and enumerators related to patterns and their management in the prototype application. A more detailed description of these objects follows.

A *ColumnDataType* enumerator serves as a generalization of proprietary data types on different databases. For example, all text-based data types, like *varchar* or *XML*, should be labeled as a *string*, and all types with decimal places like *float*, *smallmoney*, or *real* should be labeled as a *decimal.* This is important for pattern versatility. Users should not be bothered with different data types on various platforms when defining a pattern.

An abstract class *Pattern* was implemented to hold all of the common properties of previously defined patterns. This includes case sensitivity, percent of matches required, target language, and whether the pattern is enabled (active). Each of those patterns then has a dedicated class that dervives from it. Regular expressions are stored as strings in *PatternStructure*, *PatternRegex*, and *PatternLexicon* (fallback) classes. *PatternRegex* class also contains a *HashSet* of *ColumnDataTypes* as the *AlternativeDataTypes* property. As the name suggests, this can be used to specify which data types (apart from the string) the pattern should be evaluated on. The *Dictionary*-based property *SpecializedPatterns* can store one specialized pattern for each database handler.

PatternLexicon uses a string array to store the list of values. In case the database execution is not available or preferred, fallback to the previously mentioned regex property can be used instead. *PatternInteger* and *PatternDouble* classes store their range properties as *BigIntegers* to make sure that data types on any database will fit. *PatternDouble* also has range for decimal places. Finally, *PatternDate* uses the *DateTime* type for its range since C# doesn't have a dedicated type just for date.

Patterns can be grouped using *PatternGroup* classes. The *Compliance-*

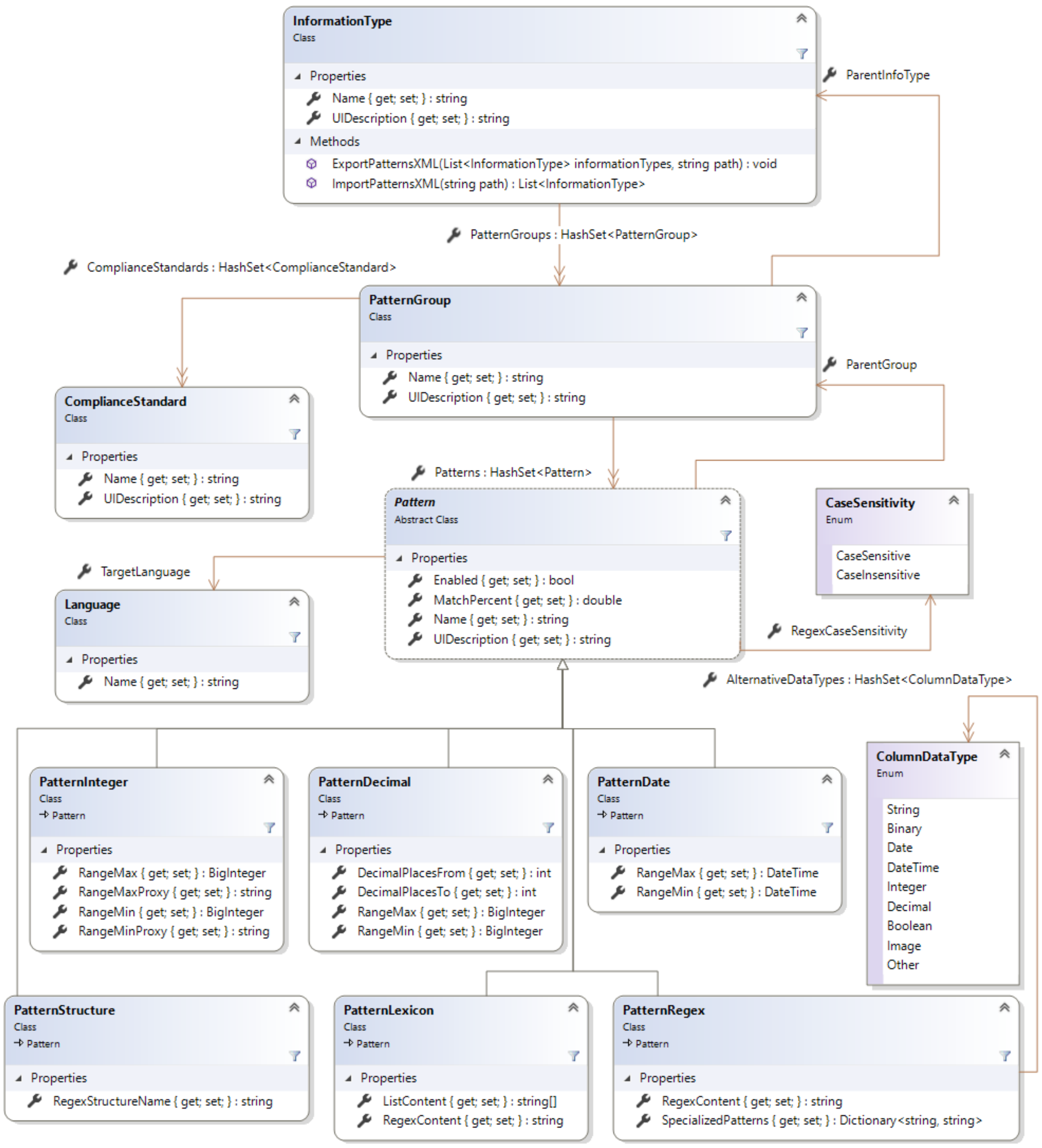
Figure 6.1: Overriding the *Equals* and *GetHashCode* methods. This ensures, that when two *PatternGroup* objects are compared in structures like the *HashSet* or the *Dictionary*, they are compared based on the property *Name*. All classes related to patterns override these method similarly.

```
public class PatternGroup
{
    ...

    public override bool Equals (object obj)
    {
        if (!(obj is PatternGroup patternGroup))
            return false;
        else if (this.Name == patternGroup.Name)
            return true;

        return false;
    }
    public override int GetHashCode ()
    {
        return System.HashCode.Combine (Name);
    }
}
```

Standards class provides the ability to assign compliance standards to these groups. *PatternGroup* stores them in a *HashSet* to ensure easy lookup. Lastly, the *InformationType* class is used to group *PatternGroups* by their usage.

All classes mentioned in this subchapter always have the *Name* and *Description* properties defined as *strings*. They also overload *Equals* and *GetHashCode* methods to guarantee uniqueness when used in a *HashSet* or *Dictionary*. These methods are used for default comparisons and are based entirely on the property *Name*, effectively making this property unique (figure 6.5).

Figure 6.2: Class diagram of patterns and related classes (some methods and properties are omitted)


InformationType
Class
Properties
Name { get; set; } : string
UIDescription { get; set; } : string
Methods
ExportPatternsXML(List<InformationType> informationTypes, string path) : void
ImportPatternsXML(string path) : List<InformationType>
ParentInfoType
PatternGroups : HashSet<PatternGroup>
ComplianceStandards : HashSet<ComplianceStandard>
PatternGroup
Class
Properties
Name { get; set; } : string
UIDescription { get; set; } : string
ParentGroup
ComplianceStandard
Class
Properties
Name { get; set; } : string
UIDescription { get; set; } : string
Patterns : HashSet<Pattern>
Pattern
Abstract Class
Properties
Enabled { get; set; } : bool
MatchPercent { get; set; } : double
Name { get; set; } : string
UIDescription { get; set; } : string
CaseSensitivity
Enum
CaseSensitive
CaseInsensitive
RegexCaseSensitivity
TargetLanguage
Language
Class
Properties
Name { get; set; } : string
AlternativeDataTypes : HashSet<ColumnDataType>
PatternInteger
Class
Pattern
Properties
RangeMax { get; set; } : BigInteger
RangeMaxProxy { get; set; } : string
RangeMin { get; set; } : BigInteger
RangeMinProxy { get; set; } : string
PatternDecimal
Class
Pattern
Properties
DecimalPlacesFrom { get; set; } : int
DecimalPlacesTo { get; set; } : int
RangeMax { get; set; } : BigInteger
RangeMin { get; set; } : BigInteger
PatternDate
Class
Pattern
Properties
RangeMax { get; set; } : DateTime
RangeMin { get; set; } : DateTime
ColumnDataType
Enum
String
Binary
Date
DateTime
Integer
Decimal
Boolean
Image
Other
PatternStructure
Class
Pattern
Properties
RegexStructureName { get; set; } : string
PatternLexicon
Class
Pattern
Properties
ListContent { get; set; } : string[]
RegexContent { get; set; } : string
PatternRegex
Class
Pattern
Properties
RegexContent { get; set; } : string
SpecializedPatterns { get; set; } : Dictionary<string, string>

6.2 Database Structure and Results

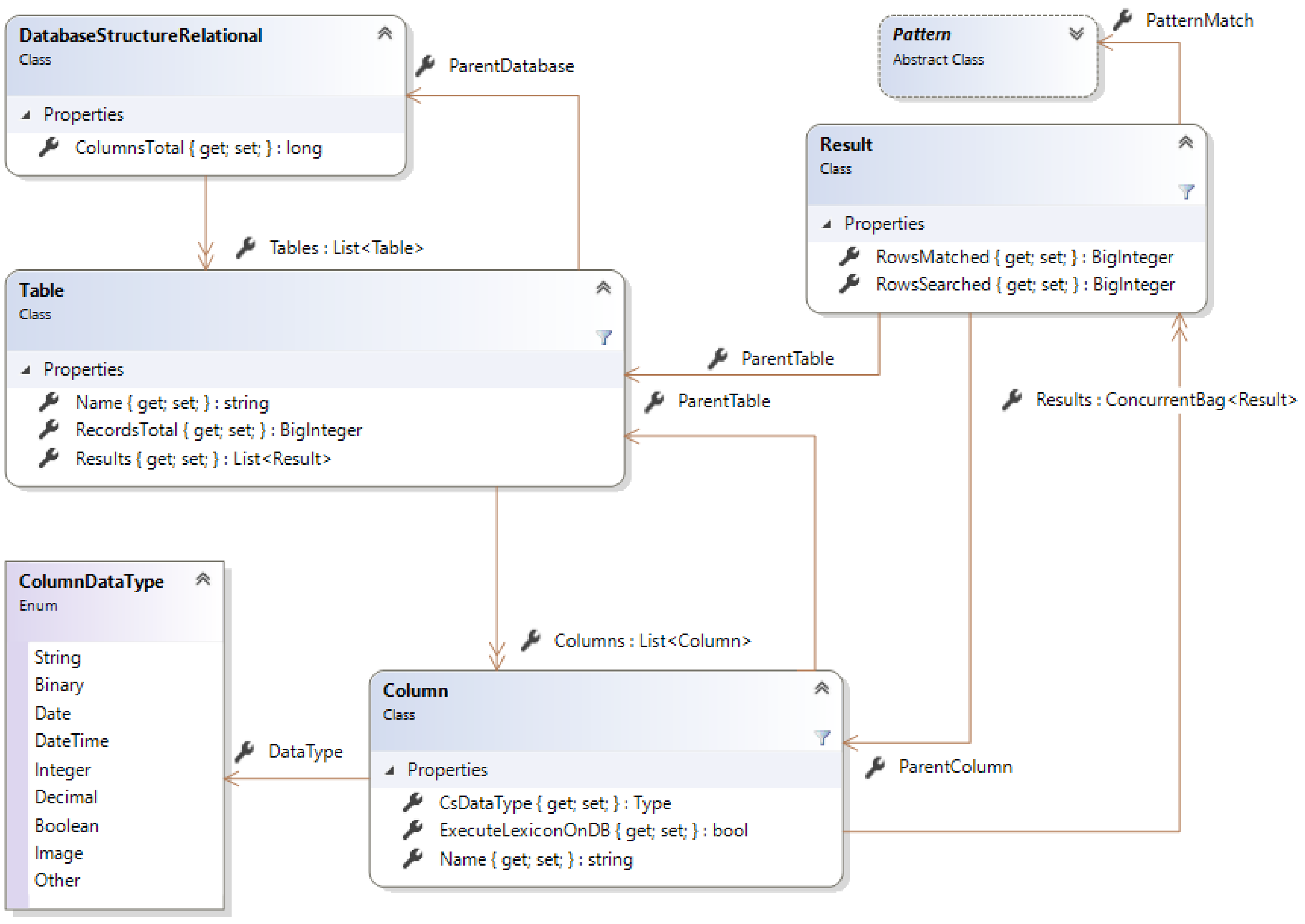
The internal database structure is implemented in step with what was suggested in the design phase. The *DatabaseStructureRelational* class contains *Table* objects in a *List*, and by analogy, the *Table* class contains a *List* of *Column* objects. It is also possible to get back to the top using properties *ParentTable* in a *Column* and *ParentDatabase* in a *Table*. For calculating the percentage of data that should be searched, the *Table* object holds the total number of rows. The *Column* class contains the previously introduced *ColumnDataType* to help the dispatcher decide what searching operations to execute. The *ExecuteLexiconDB* property can be used to flag columns, that are unfit for large-scale lexicon evaluations to be evaluated locally instead.

The *Result* class stores the search result for one pattern and one searched column or table (name). Note that it has properties *RowsMatched* and *RowsSearched*. While the *RowsSearched* property might seem redundant because the number of rows is already stored in the *Table* object, it has its purpose. Any row searched that is null should not be counted to prevent skewing the minimal required percentages. Think of a newly introduced column in an old database that is just starting to get populated with sensitive data. Since most of this column's values are nulls, patterns that define some minimal percentage of matches required could (falsely) not be recognized.

The *Column* class stores *Result* objects in a *ConcurrentBag*. This collection, in contrast with *List*, allows thread-safe addition and removal. Patterns that are being evaluated in parallel can, therefore, be easily added to the collection right away without worrying about race conditions and similar problems. Figure 6.3 shows a class diagram containing all classes mentioned in this subchapter.

Figure 6.3: Class diagram of the database structure and results (all methods and some properties are omitted)

Dispatcher


DatabaseStructureRelational
Class
Properties
ColumnsTotal { get; set; } : long
ParentDatabase
Tables : List<Table>
Table
Class
Properties
Name { get; set; } : string
RecordsTotal { get; set; } : BigInteger
Results { get; set; } : List<Result>
Pattern
Abstract Class
PatternMatch
Result
Class
Properties
RowsMatched { get; set; } : BigInteger
RowsSearched { get; set; } : BigInteger
ParentTable
ParentTable
Results : ConcurrentBag<Result>
ColumnDataType
Enum
String
Binary
Date
DateTime
Integer
Decimal
Boolean
Image
Other
DataType
Columns : List<Column>
Column
Class
Properties
CsDataType { get; set; } : Type
ExecuteLexiconOnDB { get; set; } : bool
Name { get; set; } : string
ParentColumn

6.3 Intel Hyperscan Wrapper

Since Hyperscan is a C library and no wrapper for the C# language currently exists, the author of this thesis had to write his own. The project does not release any binaries (all releases are source code only), so one must build their own. As different operating systems use different formats of dynamic (shared) libraries, at least three such libraries have to be produced. For Microsoft Windows, the output format needed is DLL (Dynamic-link library), GNU/Linux requires a shared object (commonly with .so extension), and Apple macOS uses a dynamic library (.dylib). This means that Hyperscan has to be built on these three systems and binaries produced then bundled with at least three different versions of the final product.

Figure 6.4: The hs_compile_error structure from Hyperscan's C source ported to C to be used as function argument

```
[System.Runtime.InteropServices.StructLayoutAttribute
(System.Runtime.InteropServices.LayoutKind.Sequential)]
public struct hs_compile_error
{

    /// char*
    [System.Runtime.InteropServices.MarshalAsAttribute
    (System.Runtime.InteropServices.UnmanagedType.LPStr)]
    public string message;

    /// int
    public int expression;
}
```

The wrapper uses the *DllImportAttribute* and Interop Marshalling to call functions from the library and manage the memory of the passed arguments. Some structures in the Hyperscan source, like enumerators, had to be replicated in the C# code to make method arguments more readable. No unsafe code is used in the wrapper, and all native methods are separated in the *NativeMethods* class, as per convention. Figures 1 and 2 show exemplary snippets from the implementation.

The wrapper code itself can be left unchanged on all three platforms mentioned above because the *DllImportAttribute* works with all three dynamic library formats without any changes.

In the final implementation, only a portion of all functions that Hyperscan offers are used. These are encapsulated in the C# class called *Hyperscan.* The first of these functions is *hs_valid_platform*, which has been wrapped in the *ValidArchitecture* method. As the name suggests, this method is used at the start of the application to verify CPU support (if the SSSE3 instruction set is present). Other functions, like *hs_compile_multi* and *hs_alloc_scratch* used in the *BuildDatabase* method, unsurprisingly, are used to build Hyperscan's database using patterns provided as arguments (pattern compilation). Finally, Scan encapsulates the *hs_scan* function. All errors returned as error codes are also handled by the class and converted to exceptions. All allocated memory is freed in the destructor. UTF-8 flag is set globally for all arguments passed to Hyperscan because .NET Core uses UTF-8 encoding by default.

The PInvoke Interop Assistant project [25] was used to help with creating the wrapper.

Figure 6.5: Making the hs_compile_multi function available from C# (types in the comments above the function signature are the original C types for comparison)

```
/// Return Type: hs_error_t->int
/// expressions: char**
/// flags: int*
/// ids: int*
/// elements: unsigned int
/// mode: unsigned int
/// platform: hs_platform_info_t*
/// db: hs_database_t**
/// error: hs_compile_error_t**
[System.Runtime.InteropServices.DllImportAttribute
(libhs, EntryPoint = "hs_compile_multi",
CallingConvention = CallingConvention.Cdecl)]
public static extern int hs_compile_multi (
    [In, Out, MarshalAs (UnmanagedType.LPArray,
    ArraySubType = UnmanagedType.LPStr)] string[] expressions,
    [In, Out] uint[] flags, [In, Out] uint[] ids, uint elements,
    uint mode, IntPtr platform,
    ref System.IntPtr db, [In, Out] hs_compile_error[] error);
```

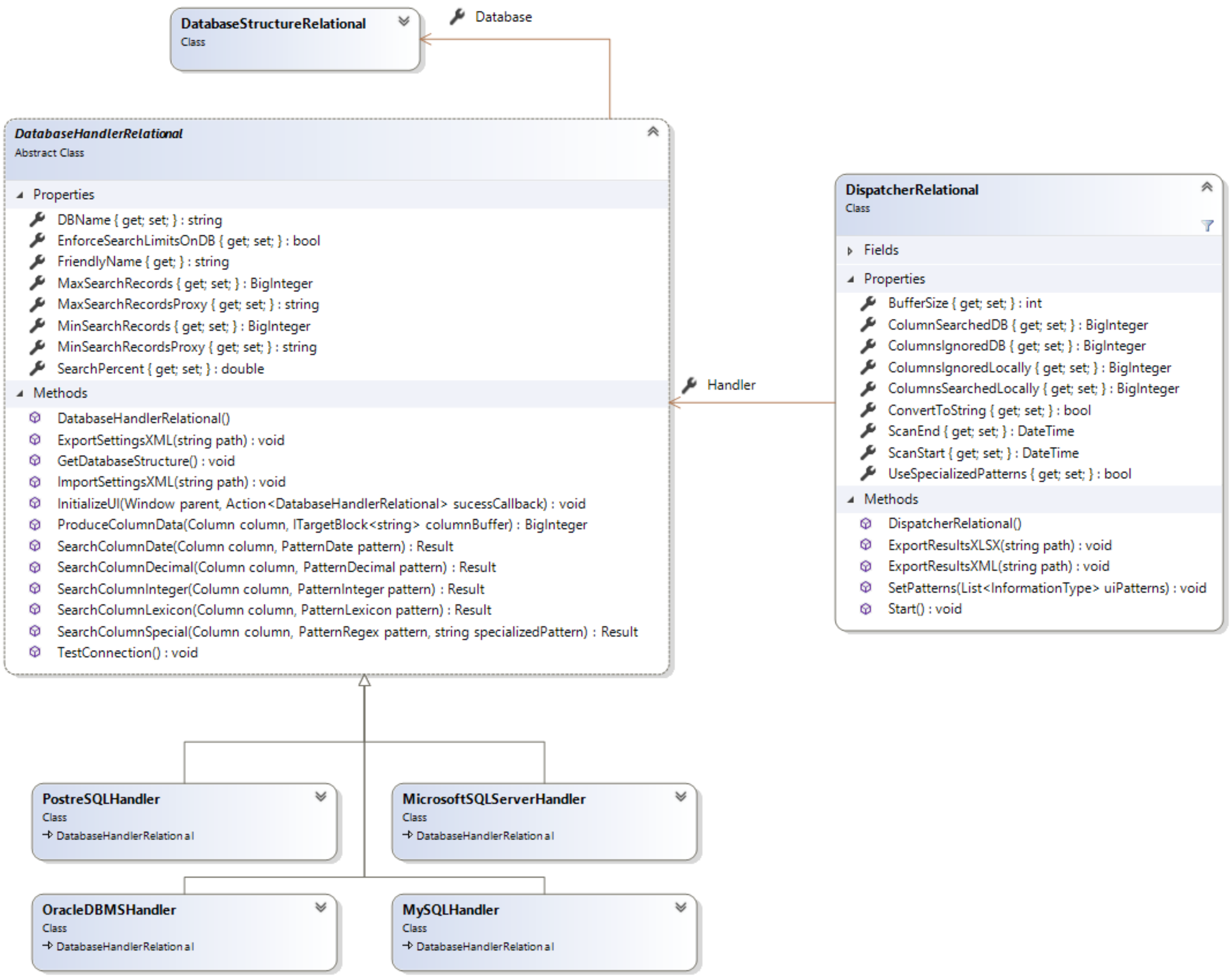
6.4 Dispatcher and Database Handlers

DatabaseHandlerRelational is an abstract class meant to incorporate all operations related to a particular database management system. All classes that perform any direct communication with a database should derive from this class and override all methods it presents. The first five of these methods (*SearchColumnSpecial*, *SearchColumnLexicon*, *SearchColumnInteger*, etc.) are used for remote evaluation. They accept the target column and pattern as arguments and are expected to handle everything necessary to return the result (or throw an exception if an error occurs). The *ProduceColumnData* method should follow the scheme (figure 4) from the design and fill the *ITargetBlock¡string¿* buffer passed in the with requested column data. Conversion to string is also left up to this method. This should most preferably be handled directly by the database, but if no such option is available, it can resort to language's ability to process the conversion.

Database structure acquisition should be managed by the method *GetDatabaseStructure*. This method fills its property *DatabaseStructureRelational* or throws an exception if something went wrong. Finally, methods *TestConnection* and *InitializeUI* (also see below) are used by user interfaces to help the user with database connection setup. Properties *MinSearchRecords*, *MaxSearchRecords*, and *SearchPercent* enable the user to control the number of rows processed or returned by any of the previous methods. All of these methods should be aware of them. If needed, this awareness can be disabled for queries evaluated directly on a database by setting the *EnforceSearchLimitsOnDB* property to false.

The *DispatcherRelational* class manages the whole database scan process and requires a *DatabaseHandlerRelational* object to be as its Handler property. Only through this handler, the dispatcher accesses any databases. As shown in figure 4, the first phase of the scan consists of discovering the database structure. For the dispatcher, this translates into calling the handler's *GetDatabaseStructure* method. The next step, pattern grouping, is handled in the *SetPatterns* method, which accepts a List of *InformationType* and is used for setting patterns for searching. These groups are then compiled into internal Hyperscan's databases, which are later used for evaluation. After the patterns are set, divided into groups, and compiled, the searching process can be started using the method Start. This method executes both remote and local evaluations in parallel. The execution can be stopped by setting the Stop property to true (since *bool* is atomic in C#, it can be used this way). Both evaluation types essentially consist of looping through all tables and evaluating patterns for each column based on its data type. In local evaluation, this also means consuming the buffer filled by the handler's *ProduceColumnData* method and forwarding this data directly to Hyperscan. By default, specialized patterns are preferred over local regex fallback, but this behavior can be changed by setting the *UseSpecializedPatterns* property to false. The last phase is the interpretation of the results, which mainly consists of filtering results based on the percentage of rows matched. Figure 6.6 shows the class diagram of the *DispatcherRelational* and *DatabaseHandlerRelational* classes with some exemplary handlers inherited.

Figure 6.6: Class diagram of the dispatcher and handlers (some methods and some properties are omitted)


DatabaseStructureRelational
Class
Database
DatabaseHandlerRelational
Abstract Class
Properties
DBName { get; set; } : string
EnforceSearchLimitsOnDB { get; set; } : bool
FriendlyName { get; } : string
MaxSearchRecords { get; set; } : BigInteger
MaxSearchRecordsProxy { get; set; } : string
MinSearchRecords { get; set; } : BigInteger
MinSearchRecordsProxy { get; set; } : string
SearchPercent { get; set; } : double
Methods
DatabaseHandlerRelational()
ExportSettingsXML(string path) : void
GetDatabaseStructure() : void
ImportSettingsXML(string path) : void
InitializeUI(Window parent, Action<DatabaseHandlerRelational> sucessCallback) : void
ProduceColumnData(Column column, ITargetBlock<string> columnBuffer) : BigInteger
SearchColumnDate(Column column, PatternDate pattern) : Result
SearchColumnDecimal(Column column, PatternDecimal pattern) : Result
SearchColumnInteger(Column column, PatternInteger pattern) : Result
SearchColumnLexicon(Column column, PatternLexicon pattern) : Result
SearchColumnSpecial(Column column, PatternRegex pattern, string specializedPattern) : Result
TestConnection() : void
Handler
DispatcherRelational
Class
Fields
Properties
BufferSize { get; set; } : int
ColumnSearchedDB { get; set; } : BigInteger
ColumnsIgnoredDB { get; set; } : BigInteger
ColumnsIgnoredLocally { get; set; } : BigInteger
ColumnsSearchedLocally { get; set; } : BigInteger
ConvertToString { get; set; } : bool
ScanEnd { get; set; } : DateTime
ScanStart { get; set; } : DateTime
UseSpecializedPatterns { get; set; } : bool
Methods
DispatcherRelational()
ExportResultsXLSX(string path) : void
ExportResultsXML(string path) : void
SetPatterns(List<InformationType> uiPatterns) : void
Start() : void
PostreSQLHandler
Class
DatabaseHandlerRelational
MicrosoftSQLServerHandler
Class
DatabaseHandlerRelational
OracleDBMSHandler
Class
DatabaseHandlerRelational
MySQLHandler
Class
DatabaseHandlerRelational

6.5 XML Serialization

Because the application uses XML in both input and output (see below), some classes require additional code. For most properties and data types, XML serialization is rather straightforward in C. Using the *XmlSerializer* class, one can serialize/deserialize many objects by calling the Serialize/Deserialize methods. In some cases, however, attributes, overloads, or proxy properties need to be added in order to resolve issues.

Figure 6.7: Proxy property for *RowsMatched*. *XmlElement* ensures consistent naming in the XML file and *EditorBrowsable (EditorBrowsableState.Never)* hides the property from IDE.

```
[XmlIgnore] //BigInteger is not serializable,
            //so a proxy property has to be used
public BigInteger RowsMatched { get; set; }

#region XML_PROXY
//Proxy property for RowsMatched
[XmlElement ("RowsMatched")]
[EditorBrowsable (EditorBrowsableState.Never)]
public string RowsMatchedProxy
{
    get { return RowsMatched.ToString (); }
    set { RowsMatched = BigInteger.Parse (value); }
}
...
```

Loading and saving an XML configuration of a database handler should be made possible by overloading the *ImportSettingsXML* and *ExportSettingsXML* methods. As the database structure isn't a setting (it is not needed for connecting to the database), it has to be labeled using the *XmlIgnore* attribute. Problems arise when attempting to serialize the BigInteger properties like *MinSearchRecords*. Since *BigInteger* doesn't implement the *IXmlSerializable* interface, serialization will return empty values. This can be resolved in two ways. The first is modifying the *BigInteger* class and making it implement the interface manually. Considering that modifying the class could cause problems later (e.g., when the language/framework is updated in the future), introducing a proxy property was perceived as a better solution. Proxy property is a property through which serialization of the underlying property is made possible while the underlying property is hidden using the *XmlIgnore* attribute. An example of a proxy property is shown in 6.7. This is used everywhere where a *BigInteger* needs to be serialized/deserialized.

Proxy property is also used to handle the serialization of the Results property. Since *ConcurrentBag* will only be serialized after it is not being accessed or modified anymore, there are no potential issues with thread safety. This means that a proxy property can be a simple *List* with appropriate getters and setters. Figure 6.9 shows the complete solution.

However, the proxy property wasn't a suitable solution for the *SpecializedPatterns* property based on the *Dictionary* collection. Instead, the well-known code for the *SeriazableDictionary* from [40] was used.

Finally, all classes that are serialized need to implement parameterless

Figure 6.8: Proxy property for *Results*

```
[XmlIgnore] //ConcurrentBag is not serializable,
            //so a proxy property has to be used
public ConcurrentBag<Result> Results { get; set; }

#region XML_PROXY
//Proxy property for Results
[XmlElement ("Results")]
[EditorBrowsable (EditorBrowsableState.Never)]
public List<Result> ResultsProxy
{
    get
    {
        List<Result> tempResults = new List<Result> ();
        foreach (var result in Results)
                tempResults.Add (result);
        return tempResults;
    }
    set
    {
        Results.Clear ();
        foreach (var result in value)
            Results.Add (result);
    }
}
#endregion XML_PROXY
```

constructors. These constructors can be made private, so they don't disrupt the design.

All pattern imports and exports from XML are made possible through *ImportPatternXML* and *ExportPatternsXML* methods of the *IntormationType* class. They serialize and deserialize a *List* of *InformationTypes* with all nested patterns and groups. This allows the XML configuration to be deserialized and passed to the dispatcher in that very form.

6.6 Database Handlers

Microsoft SQL Server was selected as an ideal candidate for a sample implementation, as it is arguably the most problematic (lack of regex support). Other handlers were created just as dummy values to demonstrate the capabilities of the UI.

6.6.1 Microsoft SQL Server

When it comes to connecting to an MSSQL Server, the .NET Framework is well equipped. Its *SqlConnection* class is optimized just for this particular database management system without generalization. Testing the connection is as simple as creating a new object with a connection string as a parameter and calling the Open method. If no exception is thrown, the connection has succeeded.

Since regular expressions aren't supported, the handler can only offer evaluation using the *LIKE* operator. Since its abilities are limited, it can be expected that local evaluation will play a more significant role. To verify whether a column contains any of the words from the lexicon pattern, at least two options can be used. The first is the *CONTAINS* operator. While being very fast, it requires the full-text index to be present for the given column. The other option is chaining the *LIKE* operator using the *OR* operators. Simple benchmarking has shown, that for columns that are not indexed, it is significantly faster to evaluate them locally rather than use the remote evaluation with the *LIKE* operator chain. For this reason, columns that aren't indexed have their *ExecuteLexiconOnDB* property set to false in the structure discovery phase. And the trouble with the *LIKE* operator isn't over, as it cannot process some text-based types like *XML*. These types have to be converted to a *nvarchar* using the *CAST* operator, which slows down the process even further. Support for other patterns (like decimal range) was implemented in a standard way without any problems, as it always consists of sending a simple query to a database and receiving the results count.

The last minor challenge was in implementing the random sample functionality correctly. The key is to avoid iterating through all of the values in a table (as even counting the rows can be time-consuming for large tables). Therefore, using conventionally recommended solutions such as *"SELECT TOP 10 PERCENT"* is not an option. Instead, one can utilize the *TABLESAMPLE* operator. Its main advantage is that it does not go through all records in a table. This is balanced out with the fact that samples on a small data (thousands of rows) are clustered based on their storage and hence not statistically sound. It also does not always return the amount of records it was requested to return precisely. Since this feature is expected to be used on enormously large databases and the number of results is known, and can be adjusted for, the author of this thesis does not consider it an obstacle.

Figure 6.9: Producing column data and filling the buffer for the MSSQL Server

```
//Init connection, build query
...
using (SqlCommand commandReadColumn
           = new SqlCommand (commandGetColumn, connection))
{
    using (SqlDataReader columnReader
           = commandReadColumn.ExecuteReader ())
    {
        while (columnReader.Read ())
        {
            //if row is null, don't count it
            if (!columnReader.IsDBNull (0))
            {
                //If DB cannot handle the conversion to string
                if (convertManually)
                {
                    object nonString = columnReader[0];
                    columnBuffer.Post (nonString.ToString ());
                }
                else
                    columnBuffer
                        .Post (columnReader.GetString (0));

                recordsProduced++;
            }
        }
    }
}
columnBuffer.Complete ();
return recordsProduced;
```

6.7 Output

While the input differs based on the way the user chooses to interact with the application, the output produced is always the same. The first output format is an XML file. After the scan is finished, the whole database structure object can be serialized thanks to the previously configured classes (figure 6.10). The purpose of the XML output is mainly machine readability so that other applications can process the results.

The other output format is an Excel (.xlsx) file. This is meant to be the primary format for the end-user. The first sheet in the file contains an overview of what database was searched, and when, how many columns were identified, what types of data were found, and so on (figure 6.14). The chart on the right side, summarizing the recognized compliance standards, was generated using the plot library ScottPlot (figure 6.13). Below this info, there is a complete list of all results. Users also have the ability to view each table's results, as each table in the database has a corresponding sheet created in the Excel file (figure 6.15). This sheet contains all columns together with their data types for the particular table. For each of these columns, a list of patterns matched, along with match percentages, is shown.

The output Excel file is generated with the help of the ClosedXML library.

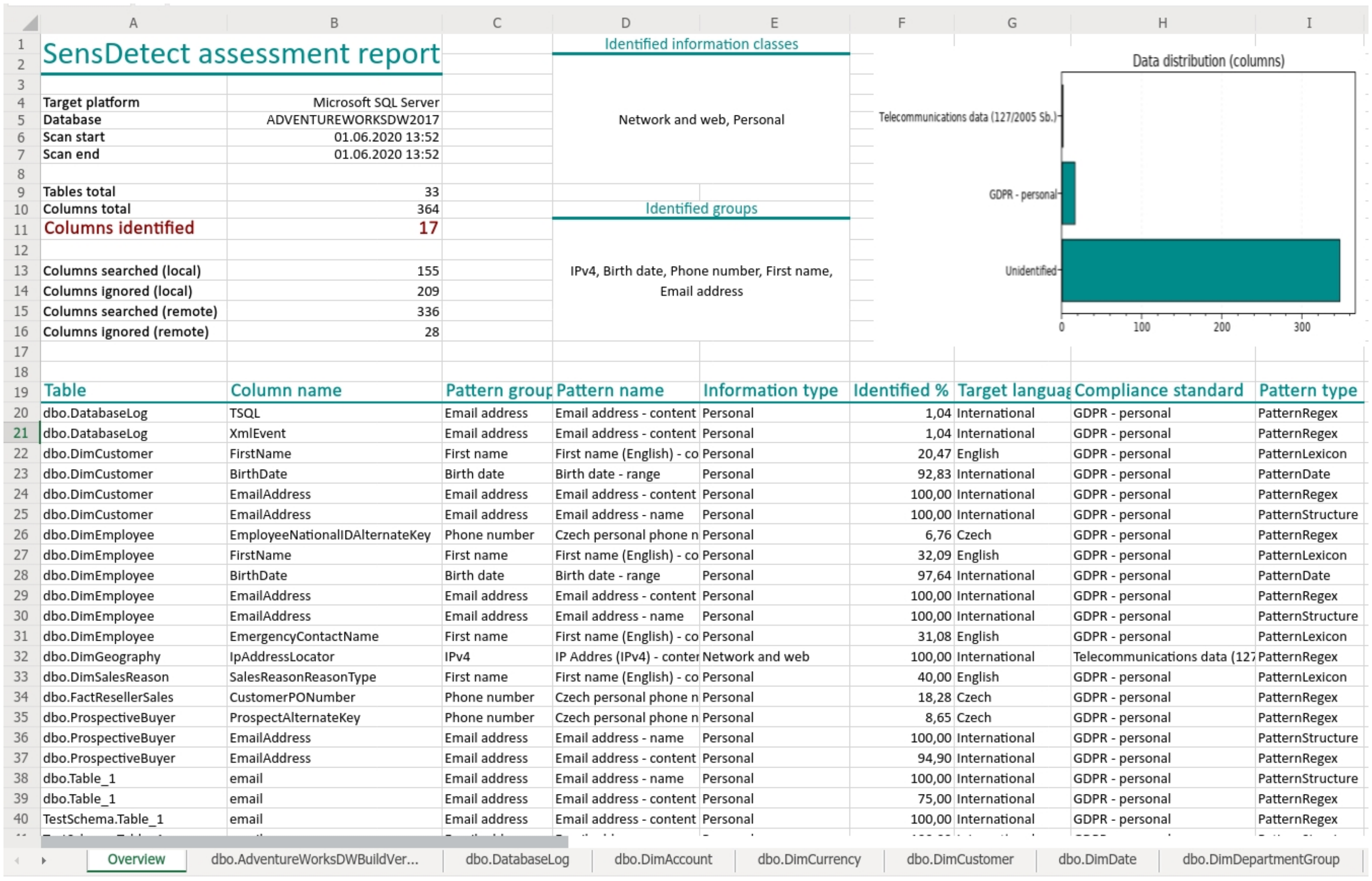
Figure 6.10: Serializing results to XML after search is finished

```
public void ExportResultsXML (string path)
{
    XmlSerializer serializer = new XmlSerializer
            (typeof (DatabaseStructureRelational));
    TextWriter writer = new StreamWriter (path);
    serializer.Serialize (writer, Handler.Database);
    writer.Close ();
}
```

Figure 6.11: Example of an XML serialized result

```
<Name>BirthDate</Name>
<DataType>Date</DataType>
<ExecuteLexcionOnDB>false</ExecuteLexcionOnDB>
<Results>
<PatternMatch xsi:type="PatternDate">
  <Name>Birth date - range</Name>
  <Enabled>true</Enabled>
  <TargetLanguage>
    <Name>International</Name>
  </TargetLanguage>
  <RegexCaseSensitivity>CaseInsensitive</RegexCaseSensitivity>
  <MatchPercent>80</MatchPercent>
  <RangeMin>1950-01-01</RangeMin>
  <RangeMax>2000-01-01</RangeMax>
</PatternMatch>
<RowsSearched>18484</RowsSearched>
<RowsMatched>17159</RowsMatched>
```

Figure 6.12: Excel output (overview)



SensDetect assessment report

Target platform	Microsoft SQL Server
Database	ADVENTUREWORKSDW2017
Scan start	01.06.2020 13:52
Scan end	01.06.2020 13:52
Tables total	33
Columns total	364
Columns identified	17
Columns searched (local)	155
Columns ignored (local)	209
Columns searched (remote)	336
Columns ignored (remote)	28

Identified information classes

Network and web, Personal

Identified groups

IPv4, Birth date, Phone number, First name, Email address

Table	Column name	Pattern group	Pattern name	Information type	Identified %	Target languag	Compliance standard	Pattern type
dbo.DatabaseLog	TSQL	Email address	Email address - content	Personal	1,04	International	GDPR - personal	PatternRegex
dbo.DatabaseLog	XmlEvent	Email address	Email address - content	Personal	1,04	International	GDPR - personal	PatternRegex
dbo.DimCustomer	FirstName	First name	First name (English) - co	Personal	20,47	English	GDPR - personal	PatternLexicon
dbo.DimCustomer	BirthDate	Birth date	Birth date - range	Personal	92,83	International	GDPR - personal	PatternDate
dbo.DimCustomer	EmailAddress	Email address	Email address - content	Personal	100,00	International	GDPR - personal	PatternRegex
dbo.DimCustomer	EmailAddress	Email address	Email address - name	Personal	100,00	International	GDPR - personal	PatternStructure
dbo.DimEmployee	EmployeeNationalIDAlternateKey	Phone number	Czech personal phone n	Personal	6,76	Czech	GDPR - personal	PatternRegex
dbo.DimEmployee	FirstName	First name	First name (English) - co	Personal	32,09	English	GDPR - personal	PatternLexicon
dbo.DimEmployee	BirthDate	Birth date	Birth date - range	Personal	97,64	International	GDPR - personal	PatternDate
dbo.DimEmployee	EmailAddress	Email address	Email address - content	Personal	100,00	International	GDPR - personal	PatternRegex
dbo.DimEmployee	EmailAddress	Email address	Email address - name	Personal	100,00	International	GDPR - personal	PatternStructure
dbo.DimEmployee	EmergencyContactName	First name	First name (English) - co	Personal	31,08	English	GDPR - personal	PatternLexicon
dbo.DimGeography	IpAddressLocator	IPv4	IP Addres (IPv4) - conter	Network and web	100,00	International	Telecommunications data (127	PatternRegex
dbo.DimSalesReason	SalesReasonReasonType	First name	First name (English) - co	Personal	40,00	English	GDPR - personal	PatternLexicon
dbo.FactResellerSales	CustomerPONumber	Phone number	Czech personal phone n	Personal	18,28	Czech	GDPR - personal	PatternRegex
dbo.ProspectiveBuyer	ProspectAlternateKey	Phone number	Czech personal phone n	Personal	8,65	Czech	GDPR - personal	PatternRegex
dbo.ProspectiveBuyer	EmailAddress	Email address	Email address - name	Personal	100,00	International	GDPR - personal	PatternStructure
dbo.ProspectiveBuyer	EmailAddress	Email address	Email address - content	Personal	94,90	International	GDPR - personal	PatternRegex
dbo.Table_1	email	Email address	Email address - name	Personal	100,00	International	GDPR - personal	PatternStructure
dbo.Table_1	email	Email address	Email address - content	Personal	75,00	International	GDPR - personal	PatternRegex
TestSchema.Table_1	email	Email address	Email address - content	Personal	100,00	International	GDPR - personal	PatternRegex

Figure 6.13: Excel output (chart)

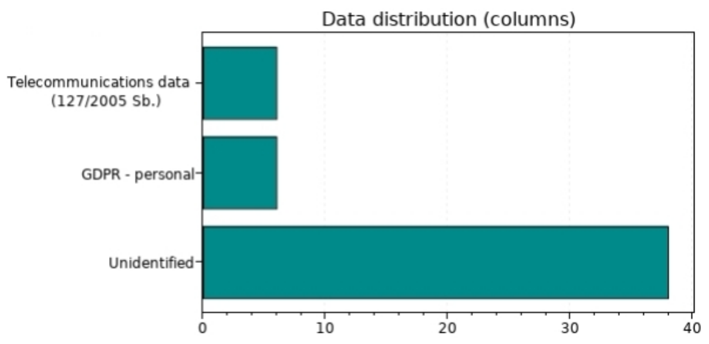
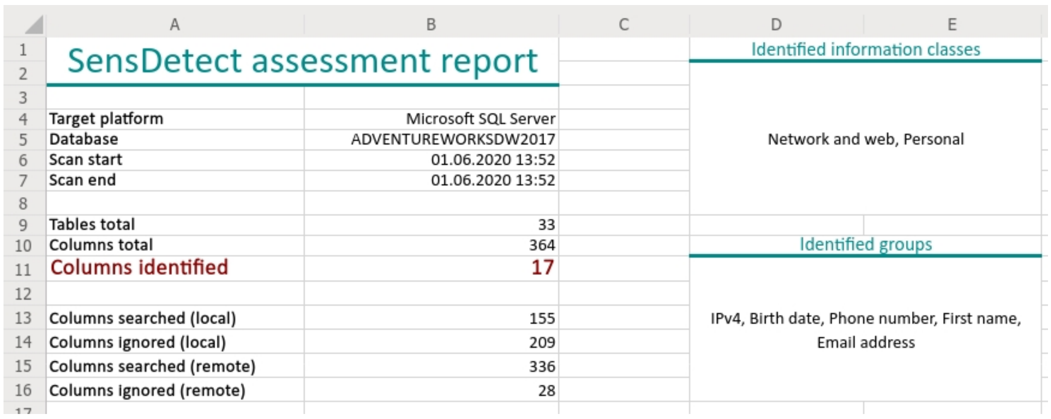
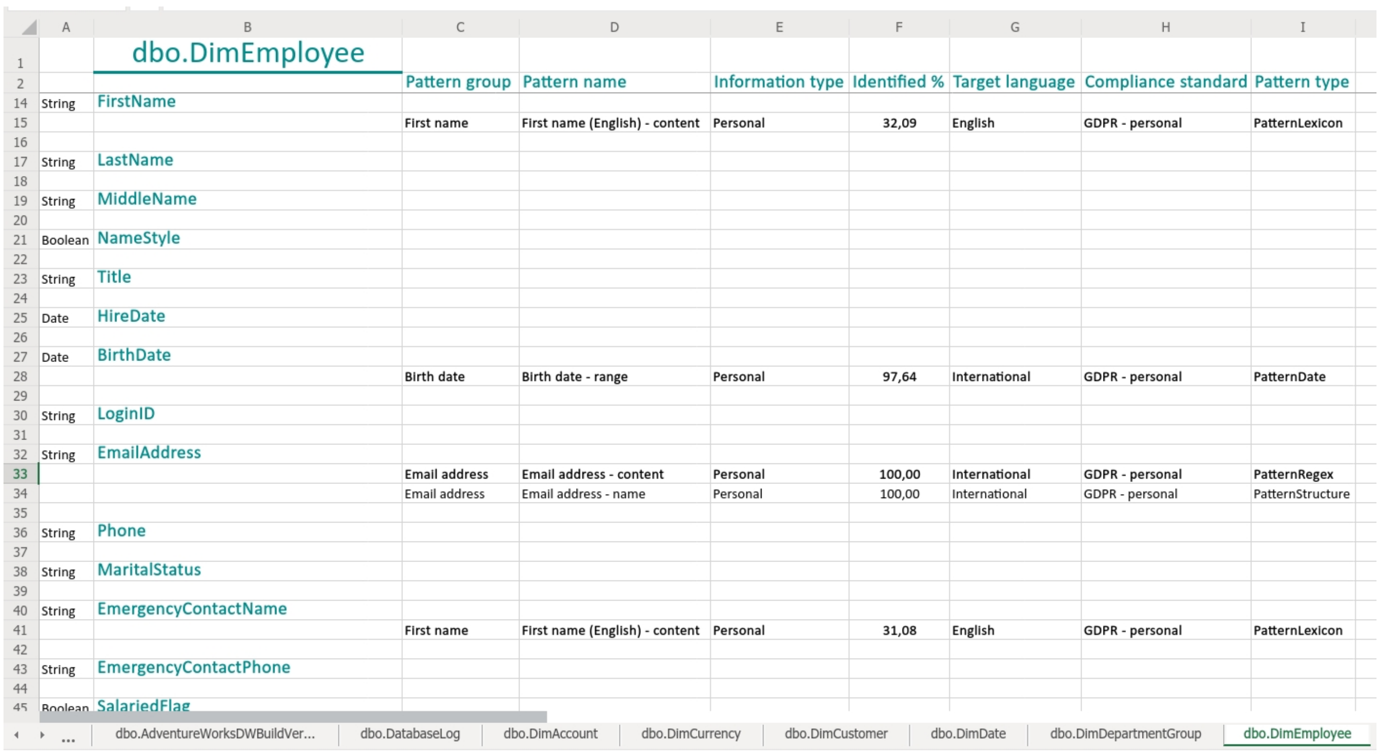


Figure 6.14: Excel output (statistics)



	A	B	C	D	E
1	SensDetect assessment report			Identified information classes	
2					
3					
4	Target platform	Microsoft SQL Server			
5	Database	ADVENTUREWORKSDW2017		Network and web, Personal	
6	Scan start	01.06.2020 13:52			
7	Scan end	01.06.2020 13:52			
8					
9	Tables total	33			
10	Columns total	364		Identified groups	
11	Columns identified	17			
12					
13	Columns searched (local)	155		IPv4, Birth date, Phone number, First name, Email address	
14	Columns ignored (local)	209			
15	Columns searched (remote)	336			
16	Columns ignored (remote)	28			

Figure 6.15: Excel output (particular table)



	A	B	C	D	E	F	G	H	I
1		dbo.DimEmployee							
2			Pattern group	Pattern name	Information type	Identified %	Target language	Compliance standard	Pattern type
14	String	FirstName							
15			First name	First name (English) - content	Personal	32,09	English	GDPR - personal	PatternLexicon
16									
17	String	LastName							
18									
19	String	MiddleName							
20									
21	Boolean	NameStyle							
22									
23	String	Title							
24									
25	Date	HireDate							
26									
27	Date	BirthDate							
28			Birth date	Birth date - range	Personal	97,64	International	GDPR - personal	PatternDate
29									
30	String	LoginID							
31									
32	String	EmailAddress							
33			Email address	Email address - content	Personal	100,00	International	GDPR - personal	PatternRegex
34			Email address	Email address - name	Personal	100,00	International	GDPR - personal	PatternStructure
35									
36	String	Phone							
37									
38	String	MaritalStatus							
39									
40	String	EmergencyContactName							
41			First name	First name (English) - content	Personal	31,08	English	GDPR - personal	PatternLexicon
42									
43	String	EmergencyContactPhone							
44									
45	Boolean	SalariedFlag							

6.8 User interface

6.8.1 Command-line

The application has a simple command-line interface. It needs two XML files as an input - patterns and handler configuration. Two previously introduced output files are produced, and their path may or may not be specified (the current working directory will be used). Figure 6.16 shows the usage example.

Figure 6.16: Command-line interface usage. First two arguments are mandatory, others are optional.

```
>SensDetect {Patterns.xml} {DBConfig.xml} [out.xml] [out.xlsx]
```

6.8.2 GUI

The main reason for introducing a graphical user interface is that creating and maintaining patterns in XML configuration files by hand is tedious and error-prone. The interface is based on the GtkSharp project and allows the user to add and edit patterns, pattern groups, and information types. These patterns can then be saved to an XML configuration file that is also accepted by the command-line version. A new scan can also be configured and started from the interface.

Figure 6.17: Obtaining friendly names of all subclasses of the DatabaseHandlerRelational class at runtime

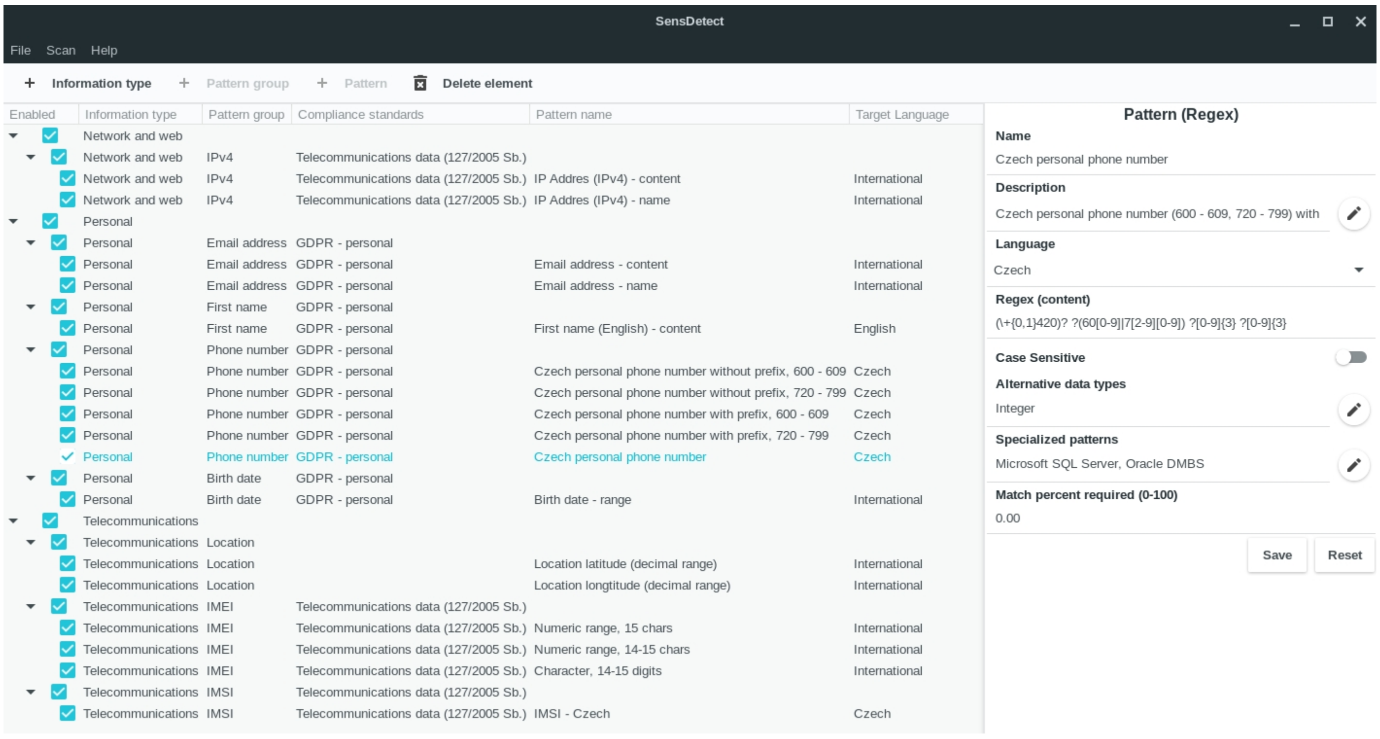
```
//Get all subclasses of DatabaseHandlerRelational
Type parentType = typeof (DatabaseHandlerRelational);
Assembly assembly = Assembly.GetExecutingAssembly ();
Type[] types = assembly.GetTypes ();
IEnumerable<Type> subclasses = types.Where
                (t => t.IsSubclassOf (parentType));

//Get all friendly names
foreach (Type type in subclasses)
{
    var instance =  (DatabaseHandlerRelational)
                    Activator.CreateInstance (type);
    PropertyInfo pi = type.GetProperty ("FriendlyName");
    var name = (string) propertyInfo.GetValue (instance);
    DatabaseTypes.Add (name, type);
}
```

The main window consists of two parts. The tree view on the left presents the user with information types, groups, and patterns, nested according to the defined structure. In it, the user can enable or disable them using checkboxes (disabled patterns are ignored during the search). When any item in the tree view is clicked, all its properties show on the right side where the user can edit them. New objects can be added using buttons at the top toolbar.

From the menu bar, patterns can be imported or exported, and a new scan can be started. When starting a new scan, the user is presented with a list of handlers. This list is obtained at runtime using reflection (code is shown in figure 6.17). This same reflection is also used when determining the list of handlers available for special patterns. When the user selects to continue, the handler's method *InitializeUI* is called. If it was not overridden, a message box with an error is shown, and the application returns back to the main window. Otherwise, the user can set up the connection to the database, select output file locations, and start the scan. Through the process of scanning, progress bars are used to help the user track the progress. If the user wishes to terminate the scan prematurely, they can click the stop button.

The following figures show parts of the interface with commentary about its usage.

Figure 6.18: Main window. This is shown right after the application is launched.



Enabled	Information type	Pattern group	Compliance standards	Pattern name	Target Language
✓	Network and web				
✓	Network and web	IPv4	Telecommunications data (127/2005 Sb.)		
✓	Network and web	IPv4	Telecommunications data (127/2005 Sb.)	IP Addres (IPv4) - content	International
✓	Network and web	IPv4	Telecommunications data (127/2005 Sb.)	IP Addres (IPv4) - name	International
✓	Personal				
✓	Personal	Email address	GDPR - personal		
✓	Personal	Email address	GDPR - personal	Email address - content	International
✓	Personal	Email address	GDPR - personal	Email address - name	International
✓	Personal	First name	GDPR - personal		
✓	Personal	First name	GDPR - personal	First name (English) - content	English
✓	Personal	Phone number	GDPR - personal		
✓	Personal	Phone number	GDPR - personal	Czech personal phone number without prefix, 600 - 609	Czech
✓	Personal	Phone number	GDPR - personal	Czech personal phone number without prefix, 720 - 799	Czech
✓	Personal	Phone number	GDPR - personal	Czech personal phone number with prefix, 600 - 609	Czech
✓	Personal	Phone number	GDPR - personal	Czech personal phone number with prefix, 720 - 799	Czech
✓	Personal	Phone number	GDPR - personal	Czech personal phone number	Czech
✓	Personal	Birth date	GDPR - personal		
✓	Personal	Birth date	GDPR - personal	Birth date - range	International
✓	Telecommunications				
✓	Telecommunications	Location			
✓	Telecommunications	Location		Location latitude (decimal range)	International
✓	Telecommunications	Location		Location longtitude (decimal range)	International
✓	Telecommunications	IMEI	Telecommunications data (127/2005 Sb.)		
✓	Telecommunications	IMEI	Telecommunications data (127/2005 Sb.)	Numeric range, 15 chars	International
✓	Telecommunications	IMEI	Telecommunications data (127/2005 Sb.)	Numeric range, 14-15 chars	International
✓	Telecommunications	IMEI	Telecommunications data (127/2005 Sb.)	Character, 14-15 digits	International
✓	Telecommunications	IMSI	Telecommunications data (127/2005 Sb.)		
✓	Telecommunications	IMSI	Telecommunications data (127/2005 Sb.)	IMSI - Czech	Czech

Pattern (Regex)

Name
Czech personal phone number

Description
Czech personal phone number (600 - 609, 720 - 799) with

Language
Czech

Regex (content)
(\+{0,1}420)? ?(60[0-9]|7[2-9][0-9]) ?[0-9]{3} ?[0-9]{3}

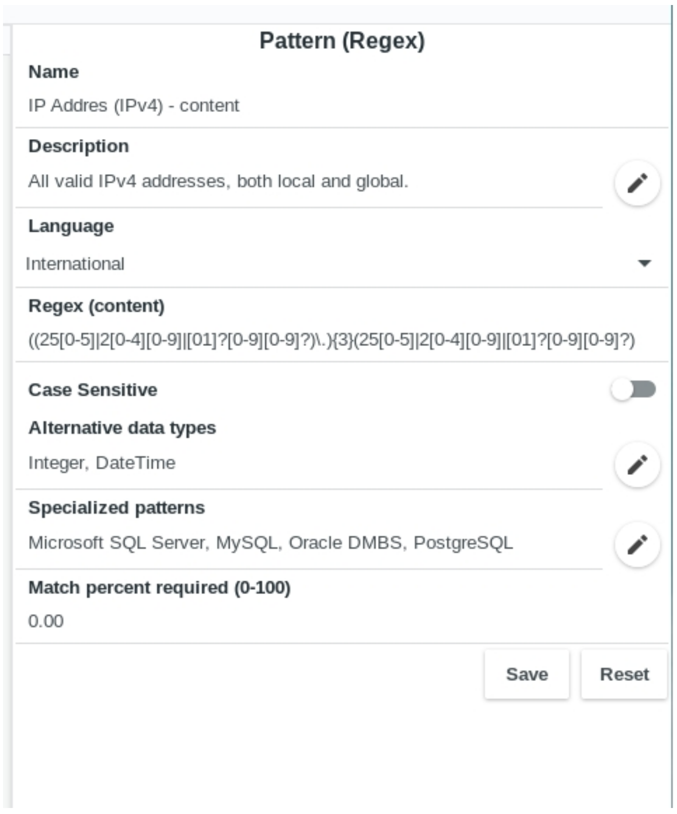
Case Sensitive

Alternative data types
Integer

Specialized patterns
Microsoft SQL Server, Oracle DMBS

Match percent required (0-100)
0.00

Figure 6.19: The right side of the main window that allows the user to view and edit selected objects from the tree view. In total, there are 8 variations of this panel.)


Pattern (Regex)

Name
IP Addres (IPv4) - content

Description
All valid IPv4 addresses, both local and global.

Language
International

Regex (content)
((25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.){3}(25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)

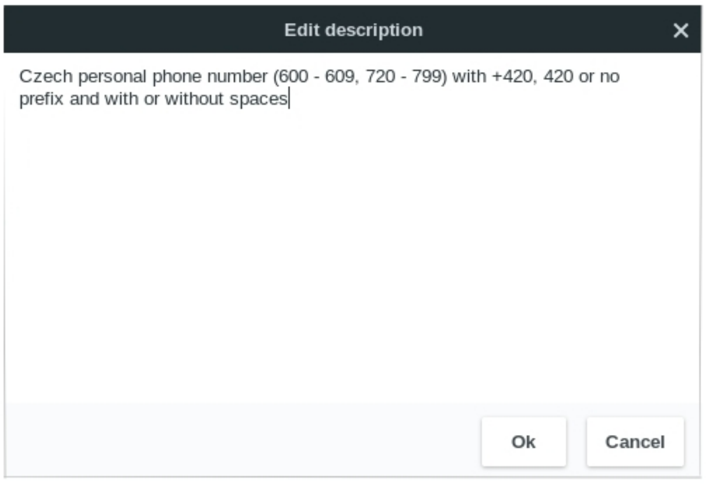
Case Sensitive

Alternative data types
Integer, DateTime

Specialized patterns
Microsoft SQL Server, MySQL, Oracle DMBS, PostgreSQL

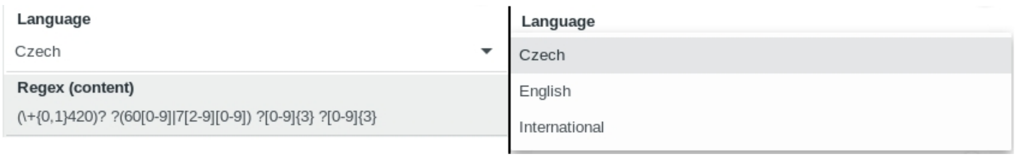
Match percent required (0-100)
0.00

Figure 6.20: Edit description dialog. Since editing a description in a small textbox can be confusing, the user can open a dedicated edit window with the button on the right side of the textbox.


Edit description

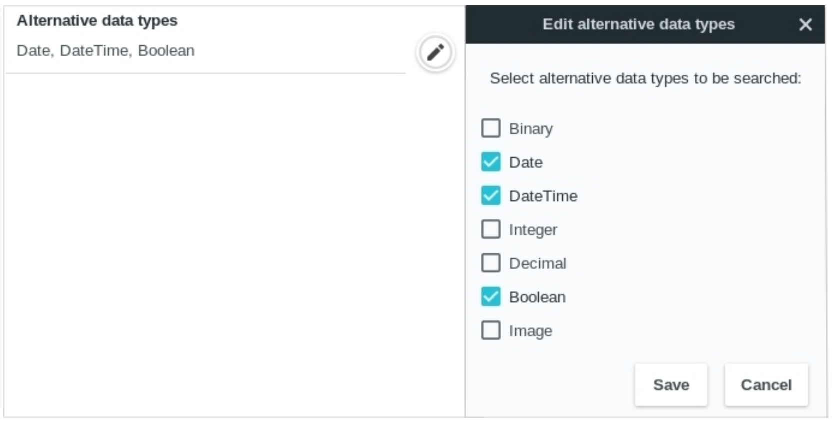
Czech personal phone number (600 - 609, 720 - 799) with +420, 420 or no prefix and with or without spaces

Figure 6.21: Combo box for setting the pattern's target language



Language
Czech
Regex (content)
(\+{0,1}420)? ?(60[0-9]|7[2-9][0-9]) ?[0-9]{3} ?[0-9]{3}

Figure 6.22: Alternative data type edit window. The user is able to add and remove alternative data types that the pattern is evaluated on. Values (checkboxes) are acquired directly from the *ColumnDataType* enumerator using reflection in runtime.



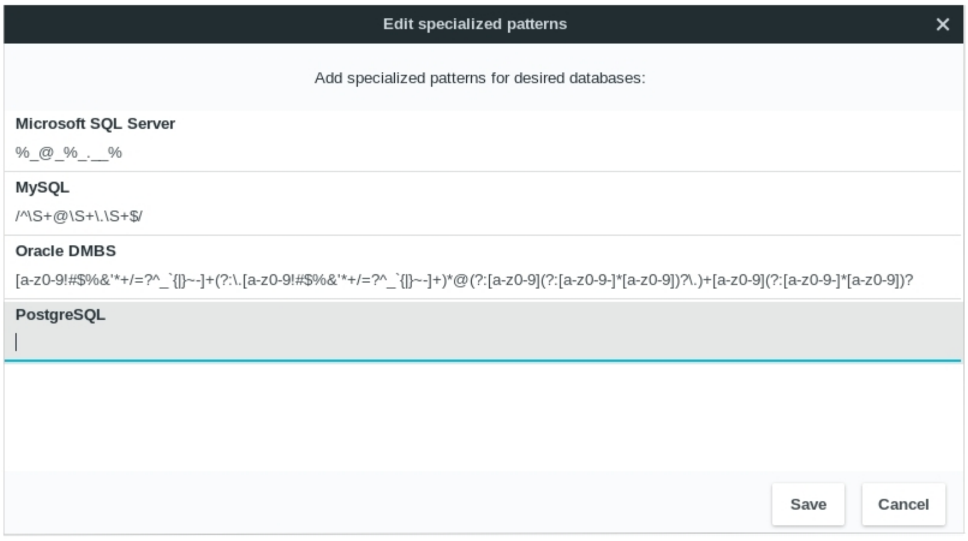
Alternative data types
Date, DateTime, Boolean

Edit alternative data types

Select alternative data types to be searched:

- [ ] Binary
- [x] Date
- [x] DateTime
- [ ] Integer
- [ ] Decimal
- [x] Boolean
- [ ] Image

Figure 6.23: Specialized patterns edit window. The list of handlers is also acquired by reflection.



Edit specialized patterns

Add specialized patterns for desired databases:

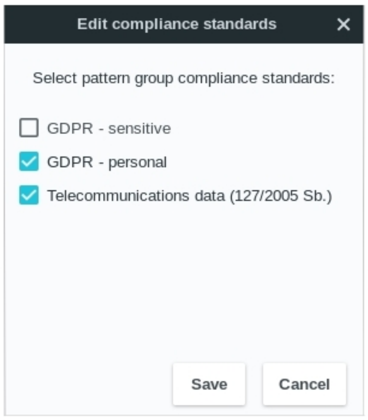
Microsoft SQL Server
%_@_%_.__%

MySQL
/^\S+@\S+\.\S+$/

Oracle DMBS
[a-z0-9!#$%&'*+/=?^_`{|}~-]+(?:\.[a-z0-9!#$%&'*+/=?^_`{|}~-]+)*@(?:[a-z0-9](?:[a-z0-9-]*[a-z0-9])?\.)+[a-z0-9](?:[a-z0-9-]*[a-z0-9])?

PostgreSQL

Figure 6.24: Edit window for compliance standards. It can be opened from the right side once a group is selected in the tree view.



Edit compliance standards

Select pattern group compliance standards:

☐ GDPR - sensitive
☑ GDPR - personal
☑ Telecommunications data (127/2005 Sb.)

Figure 6.25: Main toolbar for creating new patterns, pattern groups and information types. All of these items can also be deleted using the delete button. Button sensitivity changes based on the tree view selection.

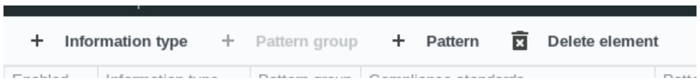
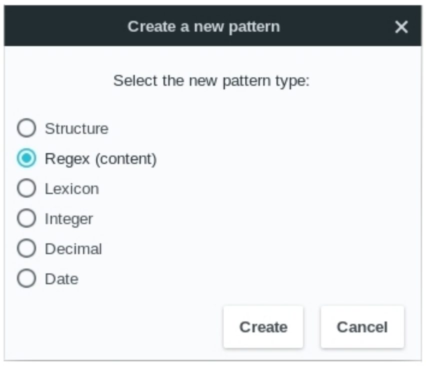


Figure 6.26: Dialog that shows when the user clicks the toolbar button to create a new pattern



Create a new pattern

Select the new pattern type:

○ Structure
● Regex (content)
○ Lexicon
○ Integer
○ Decimal
○ Date

Figure 6.27: Standards menu bar that is used to import and export patterns into XML

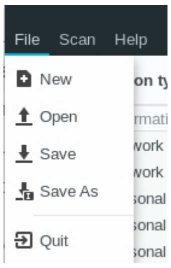
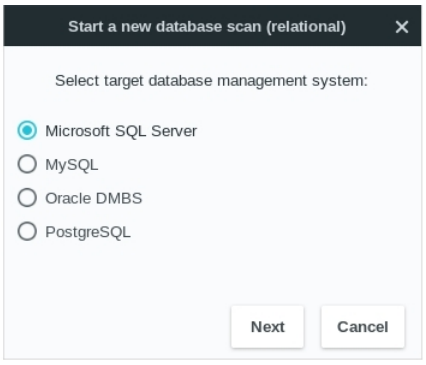


Figure 6.28: This dialog is shown when the new scan button is clicked from the menu bar. As explained in the text above, reflection is used for getting the list of available handlers,

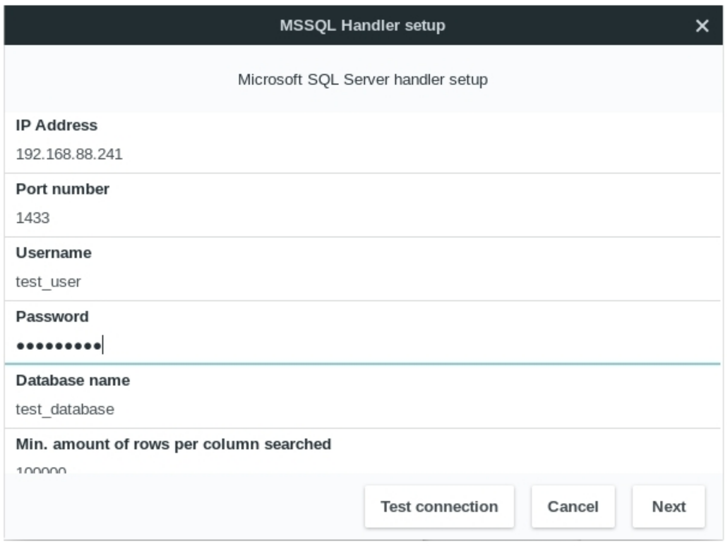


Start a new database scan (relational)

Select target database management system:

- [✓ Microsoft SQL Server]
- MySQL
- Oracle DMBS
- PostgreSQL

Figure 6.29: Database handler setup window. What is shown in this windows is based entirely on the *InitializeUI* method from the handler. All information that the handler needs for the database connection and its operation should be asked here. This includes search limits and percentages.

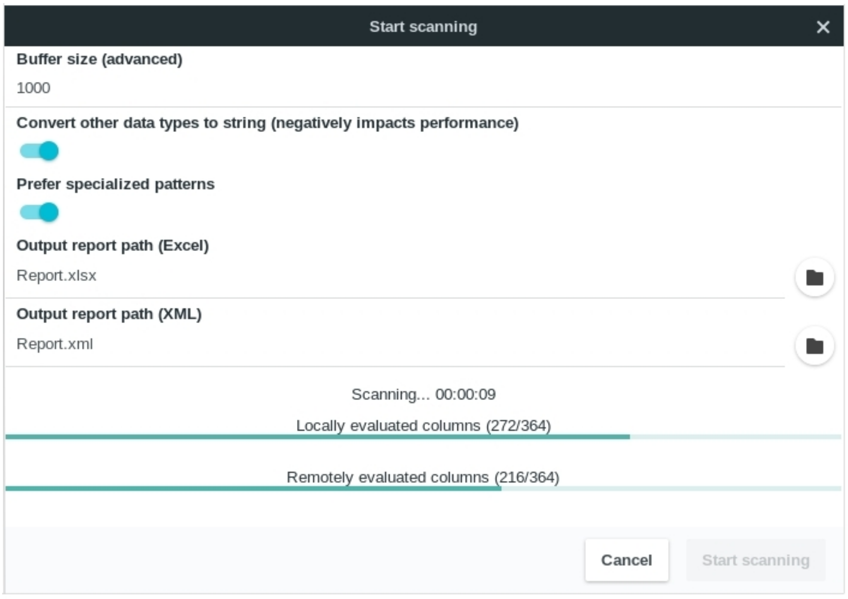


MSSQL Handler setup

Microsoft SQL Server handler setup

IP Address	192.168.88.241
Port number	1433
Username	test_user
Password	●●●●●●●●●
Database name	test_database
Min. amount of rows per column searched	100000

Figure 6.30: Finally, on the last screen, the user is able to set output paths for reports and start scanning. Patterns that require conversion of other data types to string can be disabled using the toggle button. Progress bars and labels are used to display the current progress of the search.



Start scanning

Buffer size (advanced)
1000

Convert other data types to string (negatively impacts performance)

Prefer specialized patterns

Output report path (Excel)
Report.xlsx

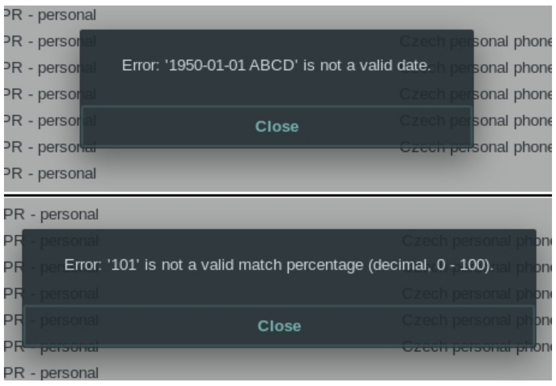
Output report path (XML)
Report.xml

Scanning... 00:00:09

Locally evaluated columns (272/364)

Remotely evaluated columns (216/364)

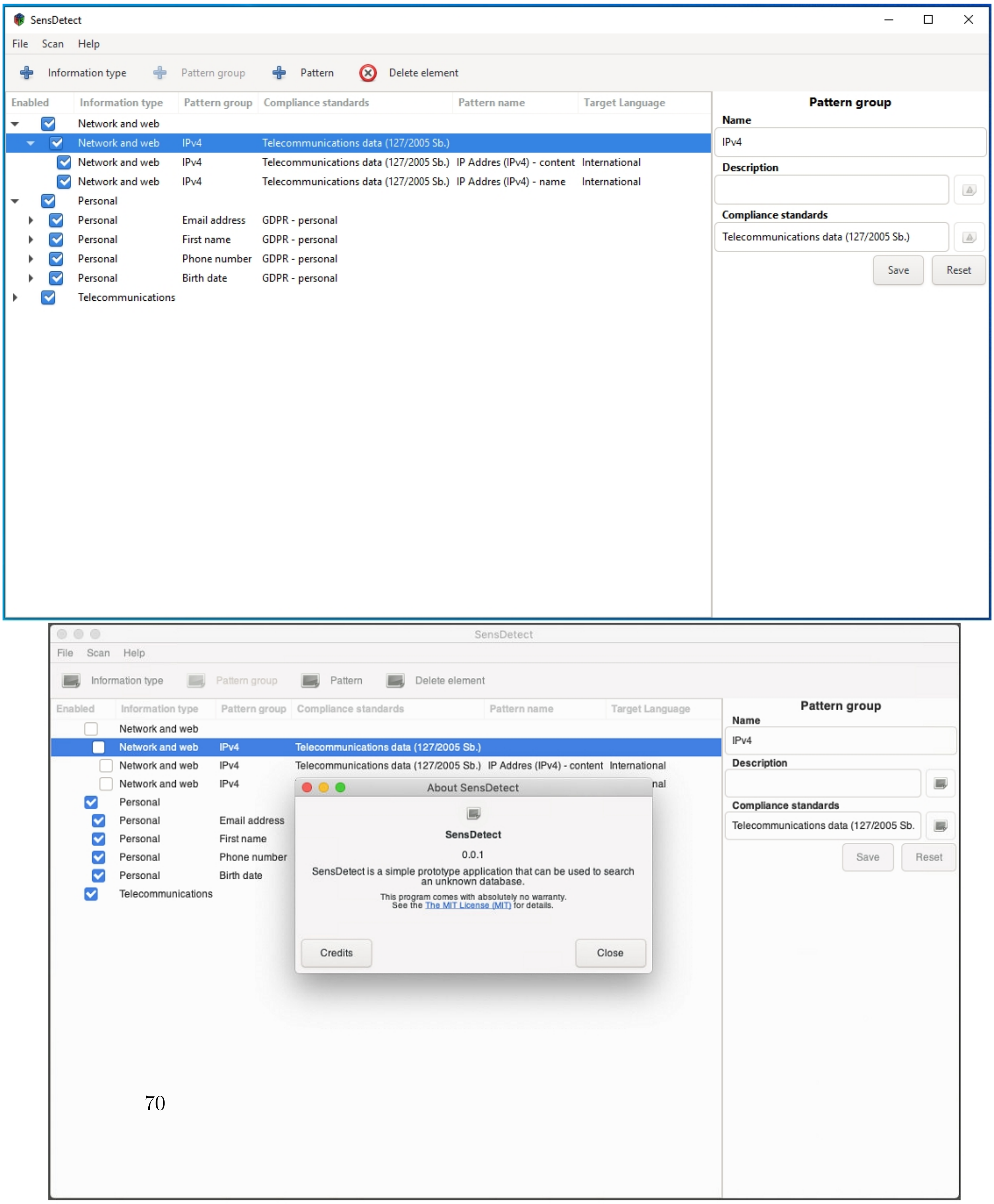
Figure 6.31: When the user attempts to save an invalid value, the application shows a message box with an error message.



Error: '1950-01-01 ABCD' is not a valid date.

Error: '101' is not a valid match percentage (decimal, 0 - 100).

Figure 6.32: Application on Windows (top) and macOS operating system (bottom). Notice one of the main disadvantages of Gtk - foreign look on other platforms



Enabled	Information type	Pattern group	Compliance standards	Pattern name	Target Language
✓	Network and web				
✓	Network and web	IPv4	Telecommunications data (127/2005 Sb.)		
✓	Network and web	IPv4	Telecommunications data (127/2005 Sb.)	IP Addres (IPv4) - content	International
✓	Network and web	IPv4	Telecommunications data (127/2005 Sb.)	IP Addres (IPv4) - name	International
✓	Personal				
✓	Personal	Email address	GDPR - personal		
✓	Personal	First name	GDPR - personal		
✓	Personal	Phone number	GDPR - personal		
✓	Personal	Birth date	GDPR - personal		
✓	Telecommunications				

Pattern group
Name: IPv4
Description:
Compliance standards: Telecommunications data (127/2005 Sb.)

Enabled	Information type	Pattern group	Compliance standards	Pattern name	Target Language
	Network and web				
	Network and web	IPv4	Telecommunications data (127/2005 Sb.)		
	Network and web	IPv4	Telecommunications data (127/2005 Sb.)	IP Addres (IPv4) - content	International
	Network and web	IPv4			nal
✓	Personal				
✓	Personal	Email address			
✓	Personal	First name			
✓	Personal	Phone number			
✓	Personal	Birth date			
✓	Telecommunications				

About SensDetect

SensDetect

0.0.1

SensDetect is a simple prototype application that can be used to search an unknown database.

This program comes with absolutely no warranty.
See the The MIT License (MIT) for details.

Pattern group
Name: IPv4
Description:
Compliance standards: Telecommunications data (127/2005 Sb.

Chapter 7

Evaluation

7.1 Performance and Detection Ability

To evaluate the performance of an application, one usually needs another application as a reference. This is a problem because other solutions that the author was able to obtain (chapter 4) either use different (unknown) methods or consider only parts of data. Therefore, any comparison drawn between these applications would be inherently biased. The author had two datasets at his disposal: the previously mentioned AdventureWorksDW2017 sample database and testing dataset by the security experts from Deutsche Telekom (see below). Even with multiple patterns defined, both were processed on a laptop in a matter of seconds, which makes any reasonable benchmarking focused on the raw performance difficult. For these reasons, no performance benchmarks were conducted.

A few testing patterns were selected to verify the application's ability to detect sensitive data; the list follows.

- **Structure**
  Email address, IPv4 address
- **Regex (content)**
  Email address, IPv4 address
- **Lexicon**
  English first name (top 30)
- **Integer**
  International Mobile Equipment Identity (IMEI), both 14 and 15 digits
- **Decimal**
  Location (both latitude and longitude)

- **Date**
  Birthdate

The first name, birthdate, IPv4, and email addresses were all identified multiple times. While the name pattern detected a few false positives, these were filtered out since at least 20 percent matches were required. IMEI and location were not detected since the database does not contain them. Figure 7.1 shows a reduced list of patterns identified with matched percentages. Both patterns and the complete results can be found in the digital attachment.

Figure 7.1: Identified columns

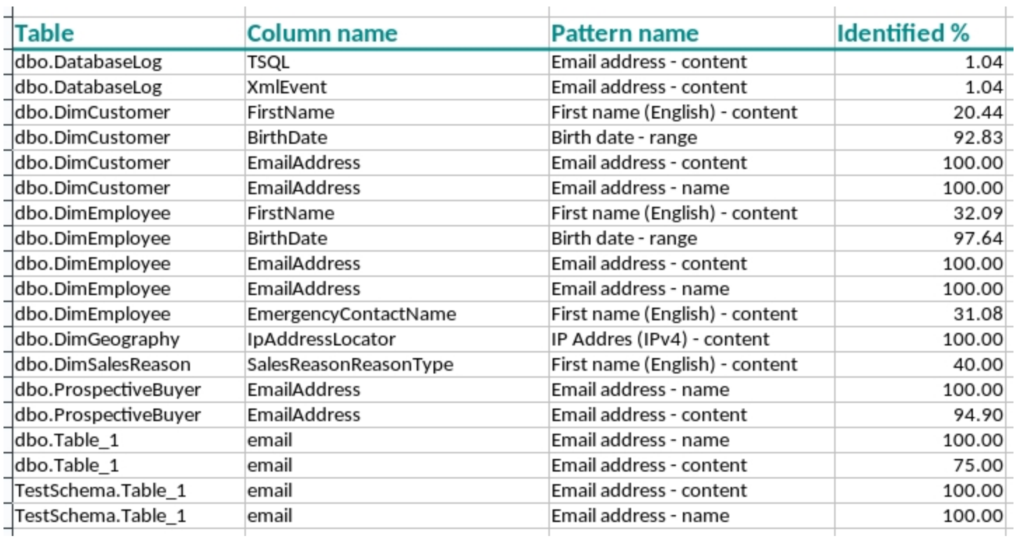


Table	Column name	Pattern name	Identified %
dbo.DatabaseLog	TSQL	Email address - content	1.04
dbo.DatabaseLog	XmlEvent	Email address - content	1.04
dbo.DimCustomer	FirstName	First name (English) - content	20.44
dbo.DimCustomer	BirthDate	Birth date - range	92.83
dbo.DimCustomer	EmailAddress	Email address - content	100.00
dbo.DimCustomer	EmailAddress	Email address - name	100.00
dbo.DimEmployee	FirstName	First name (English) - content	32.09
dbo.DimEmployee	BirthDate	Birth date - range	97.64
dbo.DimEmployee	EmailAddress	Email address - content	100.00
dbo.DimEmployee	EmailAddress	Email address - name	100.00
dbo.DimEmployee	EmergencyContactName	First name (English) - content	31.08
dbo.DimGeography	IpAddressLocator	IP Addres (IPv4) - content	100.00
dbo.DimSalesReason	SalesReasonReasonType	First name (English) - content	40.00
dbo.ProspectiveBuyer	EmailAddress	Email address - name	100.00
dbo.ProspectiveBuyer	EmailAddress	Email address - content	94.90
dbo.Table_1	email	Email address - name	100.00
dbo.Table_1	email	Email address - content	75.00
TestSchema.Table_1	email	Email address - content	100.00
TestSchema.Table_1	email	Email address - name	100.00

7.2 Modularity

The goal of the following lines is to assess how difficult it is to extend or modify the application's different parts.

- **New Pattern**
  A pattern can be added or edited in two ways. One can directly edit the configuration file and then use the file as input in both versions of the application. This file can also be edited in the graphical interface, effectively making it possible to add and modify patterns through this interface. No code needs to be changed or added.

- **New Database Support**
  Adding new database support is possible through an inherited class. A programmer needs to create a class that inherits from the *DatabaseHandlerRelational* class and overrides 11 methods to guarantee full functionality. Six methods are dedicated to pattern evaluation, one of them

being required only to fill the buffer with column data. Two methods are used to import and export connection database settings used in the command-line version of the application. One method must fill the database structure object, and the last two methods are used to test the connection and manage a GUI window. The GUI input window method is not mandatory, as the user will be automatically notified to use the command-line interface if this method is not available.

No other changes, even to the GUI, are required because reflection is used to detect new handler classes at runtime and add them in all places that require them.

It is essential to add that the above only applies to relational databases, as was discussed in the design phase. A similar architecture could be created for NoSQL databases with a dedicated dispatcher, database structure, and output.

- **New Pattern Type**
  While a new pattern type can be added by inheriting from the *Pattern* class and most of the features will work for it, there may be other issues. Each pattern type is evaluated in a different way, and that requires the code to be branched in many places. Some patterns require compilations and grouping in special structures; others can be passed to the handler without any other processing. Adding a new pattern class could, therefore, require adjustments in multiple locations in the code, depending on how the new pattern should be evaluated. The system was not built with this use case in mind.

In conclusion, it could be said that the system is modular to the extent required in the assignment and the goals set at the beginning. Adding new pattern types was not considered important in the design phase as it would likely bring small to no benefit. As discussed below, currently defined patterns types are flexible enough to detect the target data.

7.3 Expert Opinion

The application and its internals were demonstrated to security experts from Deutsche Telekom (T-Mobile), one of whom regularly conducts database audits in the production environment. The application's performance was examined on the data that they had provided explicitly for this test. Since the author of this thesis was kindly asked not to publish this data, this thesis or its digital attachment does not contain it. After the demonstration, the auditor was asked the following series of questions:

- **Can the established pattern categories cover the data defined by regulations?**
  Yes, the range of data detectable by the application is wide enough. With the right pattern combination, one is able to comply with the target regulations.

- **Is the application's design modular enough?**
  New patterns can be defined without any changes to the code, and the support for new databases can be introduced easily; thus, the application meets the requirements for modularity.

- **How would you rate the performance of the application on the testing dataset?**
  The application was able to detect all data it was meant to detect, and it did so in a reasonable time span.

- **Does the graphical user interface meet your expectations?**
  The interface appears to contain all the necessary functionalities that are required for reasonable usage.

In conclusion, it is safe to state that the application has met the expectations. However, it's important to add that more extensive testing, preferably on a production database, will be needed to evaluate the application thoroughly.

7.4 Future Work

The first thing that should be addressed before deploying the application in production is the absence of unit and integration tests. Even though they were planned to be a part of the initial release, the priorities shifted mid-development towards the unplanned GUI. Apart from conventional unit tests, the original idea was to develop a set of integration tests together with a testing database in the form of SQL queries so that it could be used in different database management systems. A programmer adding new database support could then verify their implementation with a set of integration tests. These tests would check whether particular values from the testing database were received. This would make adding new database support easier.

After the tests cover all of the code, refactoring the GUI code needs to take place. Since the author of this thesis had previous experience only with Windows-specific frameworks like WPF, he had to learn the Gtk on the go rather than understanding it comprehensively. This negatively impacted the code quality.

New GUI features should be added to make manipulation with patterns easier. The ability to filter patterns based on compliance standards and language would help to manage more extensive pattern collections. This could be added as a new item in the menu bar. Users should be alerted when leaving unsaved changes to prevent data loss. The validity of regex patterns is currently being verified only before each new scan, but it could also help to verify them right before they are saved.

Finally, the support for all kinds of NoSQL databases should be added. This would require the creation of a new dispatcher, database structure, and handler classes, as well as a more suitable output format.

Conclusion

The main goal of this thesis was to develop an application that searches an unknown database for sensitive/personal data. This, as a whole, was broken down into several separate tasks specified in the introductory chapter. The following summarizes the fulfillment of these goals.

In the first part, it was defined what the terms sensitive and personal data mean in the regulatory environment of the Czech Republic's telecommunications industry. This definition was then used to specify search patterns for this data. Various ways of crawling an unknown database were explored. Before the decision of which one to use was made, simple reverse engineering of a few commercial solutions helped to determine the best approach. Afterward, suitable technologies were selected.

Patterns can be specified using regular expressions or platform-specific queries. These patterns are then matched using either the Intel Hyperscan dynamic library or native commands executed directly on the particular database engine. The application is very modular, meaning that a developer interested in adding support for a new data source can implement an inherited class with a couple of overridden methods, and the core of the application will take care of the rest for them. The end-user is then able to add new patterns easily without any changes to the code. Finally, the performance of the application was evaluated based on available testing databases.

Bibliography

[1] *.NET documentation. How to: Implement a Producer-Consumer Dataflow Pattern.* 2017. URL: https://docs.microsoft.com/en-us/dotnet/standard/parallel-programming/how-to-implement-a-producer-consumer-dataflow-pattern (visited on 05/14/2020).

[2] *2018 Cost of a Data Breach Study: Global Overview. Benchmark research sponsored by IBM Security Independently conducted by Ponemon Institute LLC.* 2018. URL: https://www.ibm.com/account/reg/signup?formid=urx-33316 (visited on 05/04/2020).

[3] *2018 End-of-Year Data Breach Report.* 2019. URL: https://www.idtheftcenter.org/wp-content/uploads/2019/02/ITRC_2018-End-of-Year-Aftermath_FINAL_V2_combinedWEB.pdf (visited on 05/04/2020).

[4] *A comparison of regex engines.* 2017. URL: https://rust-leipzig.github.io/regex/2017/03/28/comparison-of-regex-engines/ (visited on 05/10/2020).

[5] *AdventureWorks Readme.* 2008. URL: https://github.com/microsoft/sql-server-samples/tree/master/samples/databases/adventure-works (visited on 05/04/2020).

[6] Ann Cavoukian, Stuart Shapiro, and R Jason Cronk. *Privacy engineering: Proactively embedding privacy, by design.* Office of the Information and Privacy Commissioner, 2014. URL: https://www.ipc.on.ca/wp-content/uploads/resources/pbd-priv-engineering.pdf (visited on 05/15/2020).

[7] *COMPLAINT FOR PERMANENT INJUNCTION AND OTHER RELIEF. FEDERAL TRADE COMMISSION v. EQUIFAX INC.* 2019. URL: https://www.ftc.gov/system/files/documents/cases/172_3203_equifax_complaint_7-22-19.pdf (visited on 05/04/2020).

[8] Russ Cox. *Regular Expression Matching Can Be Simple And Fast. (but is slow in Java, Perl, PHP, Python, Ruby, ...)* 2007. URL: https://swtch.com/~rsc/regexp/regexp1.html (visited on 05/09/2020).

[9] *DB-Engines Ranking.* 2020. URL: https://db-engines.com/en/ranking (visited on 05/13/2020).

[10] *Equifax to Pay $575 Million as Part of Settlement with FTC, CFPB, and States Related to 2017 Data Breach.* 2019. URL: https://www.ftc.gov/news-events/press-releases/2019/07/equifax-pay-575-million-part-settlement-ftc-cfpb-states-related (visited on 05/04/2020).

[11] *Get names of all keys in the collection.* 2018. URL: https://stackoverflow.com/a/52601873 (visited on 05/13/2020).

[12] Jan Goyvaerts. *Regular Expressions Reference.* 2020. URL: http://www.regular-expressions.info/refflavors.html?wlr=1 (visited on 05/08/2020).

[13] Graham Greenleaf. "Global data privacy laws 2017: 120 national data privacy laws, including Indonesia and Turkey". In: *Including Indonesia and Turkey (January 30, 2017)* 145 (2017), pp. 10–13.

[14] Graham Greenleaf. "Global data privacy laws: 89 countries, and accelerating". In: *privacy laws & business international report* 115 (2012).

[15] *Hyperscan 5.2 Developer's Reference Guide. Requirements, Hardware.* 2018. URL: https://intel.github.io/hyperscan/dev-reference/getting_started.html%5C#requirements (visited on 05/10/2020).

[16] *Implement INFORMATION$_S$CHEMAinHive.* 2019. URL: https://issues.apache.org/jira/browse/HIVE-1010 (visited on 05/14/2020).

[17] *Is it going to work if I try to use if with AMD processor?* 2019. URL: https://github.com/intel/hyperscan/issues/169%5C#issuecomment-509310645 (visited on 05/10/2020).

[18] *Languages Regex Benchmark.* 2020. URL: https://github.com/mariomka/regex-benchmark (visited on 05/10/2020).

[19] *MongoDB Manual. $regex.* 2020. URL: https://docs.mongodb.com/manual/reference/operator/query/regex/ (visited on 05/13/2020).

[20] *MySQL 8.0 Reference Manual. 12.7.2 Regular Expressions.* 2020. URL: https://dev.mysql.com/doc/refman/8.0/en/regexp.html (visited on 05/13/2020).

[21] *Oracle® Database Application Developer's Guide - Fundamentals. Using Regular Expressions With Oracle Database.* 2020. URL: https://docs.oracle.com/cd/B12037_01/appdev.101/b10795/adfns_re.htm%5C#1007566 (visited on 05/13/2020).

[22] THE EUROPEAN PARLIAMENT and THE COUNCIL OF THE EUROPEAN UNION. *Regulation (EU) 2016/679 of the European Parliament and of the Council of 27 April 2016 on the protection of natural persons with regard to the processing of personal data and on the free movement of such data, and repealing Directive 95/46/EC (General Data Protection Regulation) (Text with EEA relevance).* 2016. URL: `https://eur-lex.europa.eu/eli/reg/2016/679/oj`.

[23] *Pattern Matching in Search Conditions.* 2012. URL: `https://docs.microsoft.com/en-us/previous-versions/sql/sql-server-2008-r2/ms187489(v=sql.105)?redirectedfrom=MSDN` (visited on 05/13/2020).

[24] *Percentage of Individuals using the Internet.* 2019. URL: `https://www.itu.int/en/ITU-D/Statistics/Documents/statistics/2019/Individuals_Internet_2000-2018_Dec2019.xls` (visited on 05/05/2020).

[25] *PInvoke Interop Assistant.* 2019. URL: `https://github.com/jaredpar/pinvoke-interop-assistant` (visited on 05/15/2020).

[26] *PostgreSQL Documentation. 9.7. Pattern Matching.* 2020. URL: `https://www.postgresql.org/docs/9.3/functions-matching.html` (visited on 05/13/2020).

[27] Balaji Raghunathan. *The complete book of data anonymization: from planning to implementation.* CRC Press, 2013.

[28] Czech Republic. *Zákon č. 101/2000 Sb., o ochraně osobních údajů a o změně některých zákonů.* 2017. URL: `https://www.zakonyprolidi.cz/cs/2000-101` (visited on 05/15/2020).

[29] Czech Republic. *Zákon č. 110/2019 Sb., o zpracování osobních údajů).* 2019. URL: `https://www.zakonyprolidi.cz/cs/2019-110` (visited on 05/15/2020).

[30] Czech Republic. *Zákon č. 127/2005 Sb., o elektronických komunikacích a o změně některých souvisejících zákonů (zákon o elektronických komunikacích).* 2020. URL: `https://www.zakonyprolidi.cz/cs/2005-127` (visited on 05/15/2020).

[31] *Rows Across The Lake. RLIKE in hive: Filtering with regular expressions.* 2019. URL: `http://datablog.roman-halliday.com/index.php/2019/10/26/rlike-in-hive-filtering-with-regular-expressions/` (visited on 05/13/2020).

[32] LLC Securosis. "Understanding and Selecting a Data Loss Prevention Solution". In: *Securosis, LLC* (2010). URL: `https://securosis.com/assets/library/publications/DLP-Whitepaper.pdf` (visited on 05/15/2020).

[33] *Sensitive Data Discovery Solution by DataSunrise.* 2020. URL: `https://www.datasunrise.com/search-sensitive-data/` (visited on 05/05/2020).

[34] *Sensitive Data Discovery to Detect and Manage Confidential Data. Sensitive Data Detection and Protection with DataSunrise.* 2020. URL: `https://www.datasunrise.com/blog/compliances/sensitive-data-discovery/` (visited on 05/05/2020).

[35] *SQL Compliance Manager. Monitor, audit, and alert on user activity and data changes in SQL Server databases.* 2020. URL: `https://www.idera.com/productssolutions/sqlserver/sqlcompliancemanager` (visited on 05/05/2020).

[36] *SQL Data Discovery and Classification.* 2019. URL: `https://docs.microsoft.com/en-us/sql/relational-databases/security/sql-data-discovery-and-classification?view=sql-server-ver15%5C&tabs=t-sql` (visited on 05/05/2020).

[37] William Stallings. *Effective Cybersecurity: A Guide to Using Best Practices and Standards.* Addison-Wesley Professional, 2018.

[38] *Variety. A schema analyzer for MongoDB.* 2020. URL: `https://github.com/variety/variety` (visited on 05/13/2020).

[39] *What is ODBC – Open Database Connectivity. ODBC Architecture.* 2020. URL: `https://www.simba.com/resources/odbc/` (visited on 05/15/2020).

[40] *XML Serializable Generic Dictionary. Paul Welter's Weblog.* 2006. URL: `https://weblogs.asp.net/pwelter34/444961` (visited on 06/03/2020).

APPENDIX A

Acronyms

GUI Graphical user interface

UI User Interface

DLP Data Loss Prevention

API Application Programming Interface

NFA Deterministic Finite Automaton

DBMS Database Management System

SQL Structure Query Language

WPF Windows Presentation Foundation

MSSQL Microsoft SQL

Appendix B

Contents of enclosed CD

`readme.md`	the file with CD contents description
`release`	the directory with binaries
`sample_config`	the directory with sample configuration files
`MSSQL_config.xml`	MSSQL handler configuration file
`Patterns.xml`	default patterns configuration file
`scan_results`	the directory with sample scan results
`source`	the directory with the source code
`text`	the directory with the thesis text
`thesis.pdf`	the thesis text in PDF format
`thesis.zip`	LaTeX source code of the thesis (archived)