University of West Bohemia
Faculty of Applied Sciences
Department of Computer Science and Engineering

Master's thesis

Creation of Data Sources for Bibliometric Analysis

Plzeň 2019 Štěpán Baratta

Místo této strany bude zadání práce.

Declaration

I hereby declare that this master's thesis is completely my own work and that I used only the cited sources.

Plzeň, 12th May 2019

Štěpán Baratta

Abstract

The main purpose of this thesis is to create a large repository concentrating data from various publicly available databases which store bibliographic information related to intellectual property rights. One part of this work focuses on enabling access to the created repository using an application interface, providing methods for querying. In the final solution, non-relational database MongoDB was used and Java programming language was used for communication with the database. Over 200 millions of records were acquired from multiple data sources, mainly from publication database Microsoft Academic Graph. Over 3 millions of records were acquired from the United States Patent and Trademark Office. Another part of this work focused on creating an application for administering the data sources. It also enables for data preprocessing and loading data to the MongoDB database. An additional web application was created to demonstrate the functioning of the application interface, enabling for simple visualization of the results. An analysis of the main data sources was created.

Abstrakt

Cílem této práce je vytvoření rozsáhlého úložiště obsahujícího data z různých datových zdrojů, které se zabývají sběrem publikačních a patentových bibliografických dat. Součástí práce je také umožnění přístupu k této vytvořené databázi pomocí aplikačního rozhraní, které poskytuje metody pro dotazování. Ve výsledném řešení byla použita nerelační databáze MongoDB a pro komunikaci s ní byl použit programovací jazyk Java. Podařilo se shromáždit přes 200 milionů záznamů ze 4 datových zdrojů, zejména z publikační databáze Microsoft Academic Graph. Z amerického patentového úřadu United States Patent and Trademark Office bylo získáno přes 3 miliony záznamů. Další část práce se zabývala vytvořením aplikace pro administraci datových zdrojů, která také umožňuje předzpracovávání dat a jejich nahrávání do databáze MongoDB. Jako nadstavba byla vytvořena webová aplikace, demonstrující fungování aplikačního rozhraní, umožňující jednoduché vizualizace výsledků. Jako součást měření byla provedena analýza hlavních datových zdrojů.

Contents

List of Figures

Acknowledgement

I would like to thank David Budil for helping me with the preparation of some data sources.

This work was supported by project "KnowING IPR: Fostering Innovation in the Danube Region through Knowledge Engineering and IPR Management (DTP2-076-1.1) co-funded by the European Union."

1 Introduction

The main goal of the work described in this thesis is to create a large collection of data containing bibliographic information related to Intellectual property right (IPR). These involve various publication data like books, journals, articles, magazines, newspapers, reports etc. and patent data containing information about granted patents and patent applications issued around the world. In the end, this work should provide an improved access to IPR related data.

The acquired collection of data should be stored using a persistent data storage unit, like a database. Data collection will rely on extracting data from various publicly available and acquired databases, such as patent offices, publication databases (national and international) and store them in a database in appropriate format.

The outcome of this work should provide a large database mapping patent and publication data and should provide access to the acquired data using a unified API, serving for retrieval of relevant information from the primary database.

It will also provide an application to administer the data sets using a stand-alone application, from which it will be possible to manage high-level metadata of each collected data source and also provide functions to load new data to the primary database.

1.1 Outline

The thesis is separated into several chapters. The first chapter describes the background of the topic and provides an overview of the relational and non-relational data model. The second chapter provides a comparison of several non-relational databases. The third chapter describes publication and patent bibliographic databases. The fourth chapter focuses on the chosen tools and implemented software solution. Firstly, the appropriate data sources are outlined and the process of acquiring those data is described. Then, the description of data preprocessing is given. The application for data acquisition and administration is presented as a standalone desktop application. Its structure and architecture is described in detail. In the next chapter, the application providing an API to the main database is presented. An application used for demonstrating basic functioning of the API is shown. The last chapter discusses the statistics of the acquired data.

2 Database Models

2.1 Background

The term database is used to describe a variety of systems employed to organize information. The data is organized so that it is easily accessible. There are different types of database models that do this in different ways.

In this section, we will provide a brief description of the most commonly used database models and a brief comparison among those. First, we will look at traditional relational model of storing data and then we will compare it to the models that have emerged from the recent NoSQL movement based on non-relational storage models.

2.2 Relational Models versus Non-Relational Models

Relational model was first introduced by Ted Codd of IBM Research in 1970. It gained immediate attention due to its simplicity and mathematical foundation. For several decades, it has been the most common storage model, as stated by [3]. The software implementing the relational model of data is called a Relational database management system (RDBMS). The less strict term "relational database" is often used in its place.

The relational model represents the database as a collection of relations. When a **relation** is thought of as a table of values, each row in the table represents a collection of related data values. A row represents a fact usually corresponding to a real-world entity or relationship. Formally, a row is called a **tuple**, column header is called an *attribute* and the table is called a *relation*. A set of one or more attributes has to be selected to be chosen as the *primary key* which must be distinct among all tuples in the relation.

Relations can be manipulated with three basic operations: *insert*, *delete*, *update*. They can also be queried for data using a querying language. The most commonly used querying language is the "structured query language", or *SQL*.

Relational databases provide various levels of data integrity. The consistency of the database is enforced using **constraints**.

Relational integrity constraints refer to conditions which must be present for a valid relation. Constraints on the relational database can be divided

into three main categories:

- Domain constraints – Violated if an attribute value is not present in the corresponding domain or it is not of the appropriate data type.
- Key constraints – An attribute that can uniquely identify a tuple in a relation is called the primary key of the table [6]
- Referential integrity constraints – Based on the concept of foreign keys. It is an 'attribute of a relation which should be referred to from other relations. Referential integrity constraint happens where relation refers to a key atttribute of a different or same relation. However, that key element must exist in the table' [18].

Advantages of using Relational model

- Simplicity
- ACID transactional consistency – support for *joins*
- Ensured data integrity
- SQL

Limitations of using Relational model

- Cannot scale out horizontally, only vertically
- Complexity – They can become complex as the amount of data grows. The performance suffers while using a lot of *joins*
- Not good for unstructured or semi-structured data

2.3 Motivations for NoSQL

A relational approach to databases was dominant in its field for decades. Nowadays however, there is a rapid increase in a need to store huge amounts of data. The relational approach today faces serious challenges to keep up with the increasing needs in world wide web, clouds or big data applications. Applications must collect and process ever-growing volumes and varieties of data. NoSQL technologies were created to address the limitations that were experienced with regular relational databases [1].

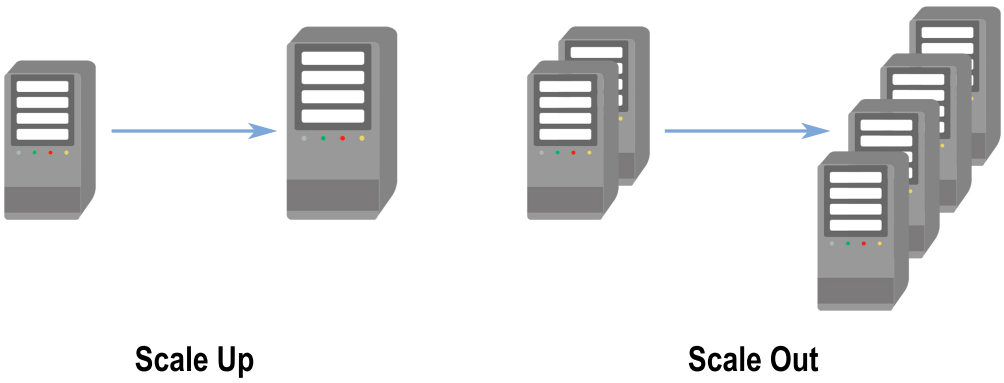


Figure 2.1: Scaling up versus scaling out

2.3.1 Scalability

It is clear that new applications have different demands than before. The main requirements on the new applications is the need to store and process more and more information. This leads to the topic of scalability.

Scalability can be handled in two ways: scaling up and scaling out. *Scale-up* architecture or vertical scaling has been a standard for storage for a long time. It refers to upgrading the resources by means of, adding more processors, memory, storage disks etc. to a single server. However this is possible up to a certain limit due to high costs and hardware limits. *Scale-out* architecture or horizontal scaling refers to increasing the resources by means of adding more servers. These can be added and removed dynamically to meet the needs of varying workloads. With relational databases, it is often very difficult to scale out and requires additional software to manage multiple servers. Oracle's *Real Applications Clusters* (RAC) is an example of cluster-based database. (see Figure 2.1).

NoSQL databases are designed to add new servers to a cluster with minimal intervention from database administrators [19].

2.3.2 Data Variety

The data that is produced today comes in different forms and varying structures. In the relational data model, everything follows a fixed, predefined schema that cannot be easily changed after it is created. This model works very well for structured data with a fixed, unified format, but does not align well when working with unstructured or semi-structured data. This is data that does not have a fixed number of fields. Data can also change structure over time. Because of these reasons, usage of relational data model for storing unstructured or semi-structured data becomes rigid and inflexible to change.

Sometimes, data can be structured, but fields for some data records may be missing. In relational database, it can be solved using "sparse data", where missing fields are filled with blank values. The disadvantage of this approach is that we store redundant data, therefore increasing the size of the database.

2.3.3 Cost

The cost is one of the essential considerations for any business organization. Commercial RDBMS software provide many licencing options which include charges for the size of the server running the RDBMS or the number of users concurrently accessing the database.

Most major NoSQL databases are usually open source and do not charge fees for running their software, so the software is free to use on as many servers as needed. Therefore NoSQL have the advantage of avoiding these issues.

2.4 Non-Relational Models

Due to the described limitations of the relational data model, many companies today are transitioning to a non-relational one. They are collectively known as a *NoSQL* databases. The term NoSQL stands for "Not only SQL", implying that it is an alternative to a traditional relational database.

The main characteristics of NoSQL database are the following:

- High scalability
- High availability
- High performance
- Schema-less
- Free of joins

There are four major types of NoSQL databases:

- Key-value databases – Every item in the database is stored as an attribute name (*key*) together with its value. The most prominent examples are Riak, Voldemort, Redis. [10]

- Document databases – Each key is associated with a complex data structure known as document. Every document can contain various fields and can also have nested documents. The most popular document-based database is MongoDB.

  The data in documents are encoded in some standard format. These formats include JSON or BSON, XML, YAML.

- Column-based databases – Stores data into a collection of columns. Provide some of the functionality of relational databases, such as the ability to link or join tables. Offer high performance for queries on large datasets. The most popular are Cassandra or HBase.

- Graph databases – The data is represented by a graph-like structure, with the nodes of the graph being the objects and their set of attributes, and the edges representing the links between those objects [14].

3 NoSQL Databases Comparison

This chapter will give basic overview of some non-relational NoSQL databases and provide a brief comparison among them. It will also discuss the advantages and disadvantages of each presented database.

3.1 MongoDB

MongoDB is the most popular non-relational database on the market today. Most prominent customers are Adobe, AstraZeneca, Barclays, eBay, Foursquare, IBM, Under Armour, Verizon Wireless and others. [27]. It is free and open-source and all versions released after October 16, 2018 are published under the *Server Side Public Licence (SSPL)v1* [25].

Data Model

Mongo stores data in a form of JSON documents (although internally, the data is stored in a BSON format which is a binary form of JSON). A Mongo document can be likened to a row in a relational database. The table 3.1 shows the relationships between Mongo and traditional relational database.

MongoDB	**Relational DB**
Database	Database
Collection	Table
Key	Column
Value	Value
Document	Record

Table 3.1: Relationship of naming common database structures between MongoDB and traditional relational database

Schema

Whereas every relational database contains a description of the structure of the data that is stored there. So when we need to modify a column for example, we need to specify which data type the column is supposed to hold.

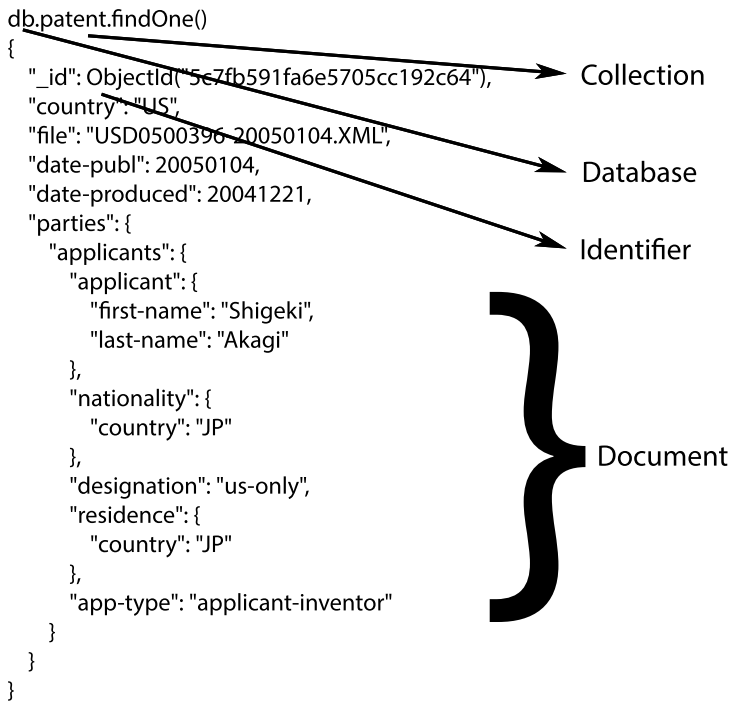


Figure 3.1: A structure of a Mongo Document

Because MongoDB is a document-based database, it does not know the concept of columns, but is based on documents, therefore the data stored in there can have varying structure which is called a schema-less structure. All that this means in the end is that schema is dynamically typed when the data is inserted to Mongo.

Accessibility

The MongoDB provides a mongo shell to interact with databases. It can be used to query or update the data or perform some administrative operations [26].

As well as access through the command line interface provided by mongo shell, MongoDB adds support for many programming languages through application drivers. They are available for many popular programming languages, like C, C++, C#, Java, PHP, Python, etc.

Indexing

Indexing is a way to optimize performance of a database by minimizing the number of disk accesses required when a query is processed.

In MongoDB, indexing is one of the most significant features when it comes to query optimization and performance tuning. If the index is present for the query, it limits the number of document that have to be scanned, therefore increases performance. MongoDB uses B-tree as a data structure to store the indexes.

There are several types of indexes in MongoDB. The most commonly used are the following:

- Single field index – A single field from the Mongo document can be indexed. It can be either ascending, specified by 1 or descending index, specified by -1.

  In mongo shell, the single field index is created using the following command:

  ```
  db.collection.createIndex( { name: 1 } )
  ```

- Compound index – Multiple fields can be defined for indexing.

  For compound indexes, mongo shell uses the following syntax:

  ```
  db.collection.createIndex({name: 1, score: -1})
  ```

- Text index – An index that supports searching for string content, thus providing basic full-text capabilities. The important note is that a collection can have at most one text index [28].

  Text index is created using the following command:

  ```
  db.collection.createIndex({abstract: "text"})
  ```

Querying

MongoDB provides its own, fully-featured querying language with syntax similar to Javascript's dot notation for arrays. The query content is constructed using the JSON format.

For example, the following is an example of a query in MongoDB:

```
db.publication.find(
{
  author: "Tolkien",
  year: {
    $gt: 1999
  }
})
```

This query searches the publication collections and returns all documents that contain "Tolkien" in the *author* field and the *year* field is greater than *1999*. This corresponds to the following SQL query:

```
SELECT * FROM publication
WHERE author = "Tolkien" AND year > 1999
```

Summary

MongoDB is the leading document-based NoSQL database on the market today. It provides a rich support for querying and fast response times. It is dynamically typed, making it very flexible. MongoDB can be easily scaled out.

Advantages

- Schema-less – As a schema-less database, every document can have different structure. This makes it a very flexible database.
- Complex querying support
- Secure
- Scalable – MongoDB is easily horizontally scalable and can be distributed onto several machines.
- Performance – If indexes are used properly, the query response is very fast in comparison to relational models.

Disadvantages

- Not intended for full-text searching
- Not suitable for relational modeling
- High memory usage

3.2 Cassandra

Cassandra is the most widely used column-based NoSQL database today. It is used in many world organizations and companies like Apple, Netflix, Uber, ING, McDonalds etc. It is licensed under the Apache License. It is known

to be very robust, providing very high availability and scalability [22], [9]. The implementation of Cassandra is done in Java programming language.

Data Model

Cassandra was mainly designed to handle huge amounts of unstructured data. It uses column families which are similar to tables in RDBMS, they contain rows and columns. The difference is that rows in a table do not necessarily need to have the same columns. The columns can be added or removed while the database is running.

As an advantage to relational databases, tables can be created, altered or dropped while the database is running or processing queries [16].

The table 3.2 shows the relationship between Cassandra and traditional relational database.

Cassandra	**Relational DB**
Keyspace	Database
Table (Column family)	Table
Cell	Column
Value	Value
	Record

Table 3.2: Relationship of naming common database structures between Cassandra and traditional relational database

One of the most important things in Cassandra is the primary key. In traditional relational database, for an effective model, it is necessary to have a lot of relationships between tables using foreign keys and relational constraints. This is not possible in Cassandra, as it is not able to model relationships and only relies on primary keys. That also means that there is less complexity in the design of the model. On the other side, it is necessary to know much more about the queries we want to run ahead of time.

There are several types of keys in Cassandra:

- Primary key – A column uniquely identifying a record in a table.
- Composite key – Primary key which is comprised of multiple columns.
- Partition key – Internally, data are identified using unique keys called row keys. They are used to partition data, so they are called partition keys. A table with single column primary key is also a partition key. A table with multiple column primary key, the first term is the partition key.

- Clustering key – Tells Cassandra, how the data is clustered.
- Composite partition key

The following figure 3.2 shows the data model of Cassandra visually.

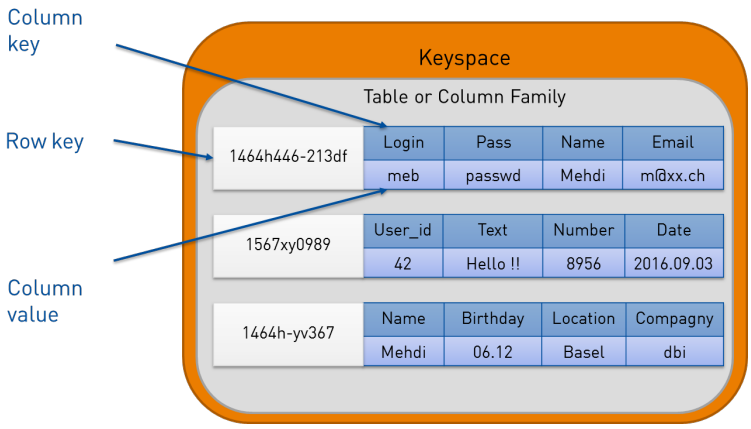


Figure 3.2: Data model of Cassandra database

Schema

Cassandra, in contrast to MongoDB for example, is not exactly schema-free, but under the hood, it contains a lookup key for every data record in the form of primary key.

Accessibility

Cassandra provides a query language shell called *cqlsh* that allows users to communicate with the database using its own query language Cassandra Query Language (CQL).

Also, there are many client drivers for most popular programming languages, like Java, Python, Ruby, C#, C++ etc.

Querying

Similarly to MongoDB, Cassandra uses its own querying language. It is called CQL and its syntax is similar to SQL's syntax. Using *cqlsh*, the user is able to create keyspaces and tables, perform insertions and query tables.

3.3 ElasticSearch

ElasticSearch or simply Elastic is described as a 'real-time distributed search and analytics engine'. It is built upon Apache Lucene which is a very complex library allowing for full-text searching. [4]. Elastic uses Lucene for indexing and searching, but tries to hide all the complexities of the library. It accomplishes that by providing a simple RESTful API, easily accessible by users.

As well as a search engine, Elastic serves as full scale distributed database able to store documents where every field is indexed and searchable. Similarly to MongoDB, it uses JSON format to store the documents.

ElasticSearch and Lucene are all written in Java programming language.

Data Model

ElasticSearch is a document-oriented database. That means that it does not store rows and columns like in a relational database, but stores whole objects (documents). Elastic is able to store them and index their contents using indexes to make them searchable.

Instead of a regular index, Elastic uses a structure called *inverted index*. It contains a 'list of all the unique words that appear in the document, and for each word, a list of the documents in which it appears in'[23].

For example, if we have two documents, each with a field *content* with the following texts:

1. doc_1 – The dog was found by the owner.
2. doc_2 – The owner looked for his dogs.

Elastic creates the inverted index by splitting the text into separate words (terms). For each unique term, it keeps track of the documents it appeared in. The terms are not in fact stored like this, but rather in their normalized form. That includes stemming, lemmatization and other techniques that will not be described here.

The final inverted index can look something like this:

Term	doc_1	doc_2
the	x	x
dog	x	x
was	x	
found	x	x
by	x	
owner	x	x
looked		x
for		x
his		x

If we want to search for the text `dog looked`, we need to look for the documents, in which each term appears:

Term	doc_1	doc_2
dog	x	x
looked		x

We can see that the first document matched in more terms from the query than the second one. Therefore the first document is a better match.

Accessibility

All the functionality is provided using a RESTful API, so it can be accessed using a web client or a command line. It also provides drivers for most programming languages.

Comparing to MongoDB

MongoDB and ElasticSearch can seem very similar at first glance, but in fact, the primary usage of both databases differs quite drastically. MongoDB is usually best serves as a general purpose database, while ElasticSearch is used mainly for its full-text searching capabilities as a standalone search engine, but it is not so good for inserting new data.

Because ElasticSearch is not primarily thought of as a database, but rather a search engine, it is often used together with other NoSQL or even SQL databases for persistent storage.

Another difference between MongoDB and Elastic is the way they handle indexes. While Elastic uses a structure called inverted index, MongoDB's indexes are based on traditional B+ tree. Elastic's default behavior is to index every field of a document, while in MongoDB, we have to define indexes

on specified fields and they are not as complex and the number of indexes which can be present on the collection is limited.

While MongoDB is implemented in C++, ElasticSearch's implementation is in Java.

3.4 Graph Database – Neo4J

Neo4J is an open source, number one rated graph database [36]. It is used to store, query and analyze highly connected data. Its key customers include eBay, Walmart, Cisco, Airbus, Verizon, US Army and many others.

The following paragraphs were created with support of [21].

Neo4J is an ACID-compliant database and is intended for an Online transaction processing (OLTP). However that does not mean that it cannot be used for analytical tasks, though it is not optimized for them. It is also designed to be highly scalable.

Data Model

In a graph database, data is modeled differently from a relational database. Modeling the data in a graph database can be seen as an advantage, as it is more natural then data modeling in a relational database. It is mainly because it can be more understandable even for non-technical people.

For example, we can have two entities, a *Patent* and an *Author* entity and we want to create a relationship between these two entities which have a n:m identifying relationship. An example of an entity relationship model containing these two entities in a relational database is shown on Figure 3.3.

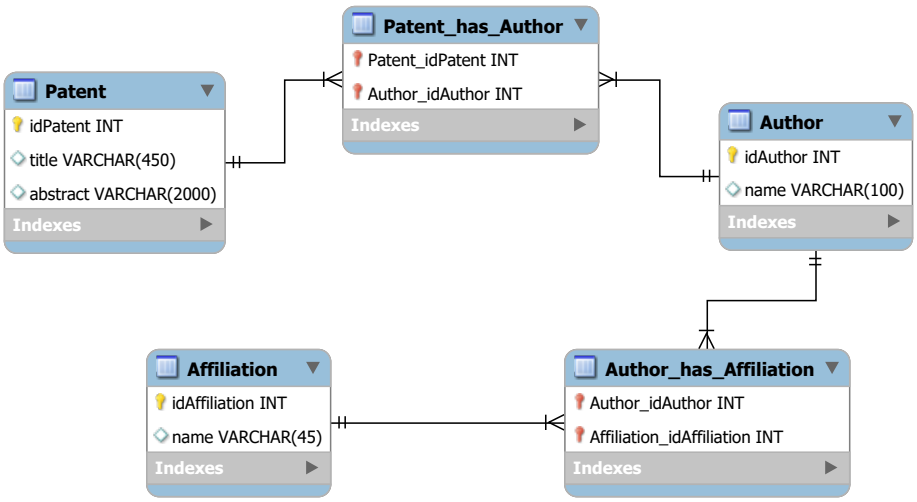


Figure 3.3: Relational data model

In a graph database, the data are modeled as a set of *vertices* and *edges* (in databases usually called *nodes* and *relationships*). They represent the data in a more natural way. The example modeled on Figure 3.3 can be modeled in a graph database similarly to the model shown on Figure 3.4. [21]

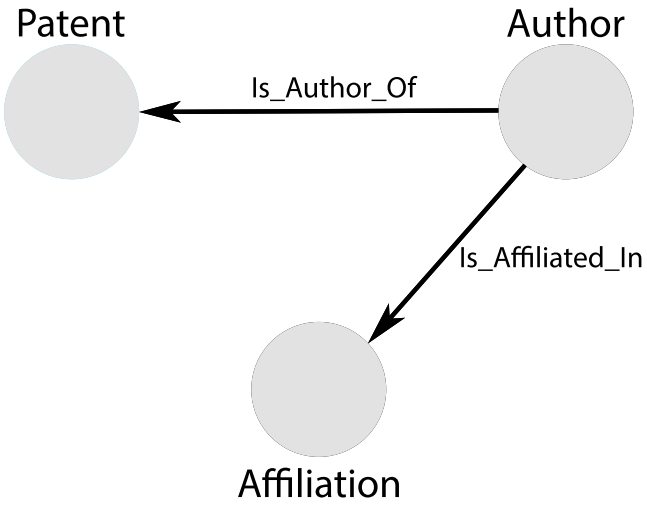


Figure 3.4: Graph data model

Querying

Neo4J defines its own querying language called *Cypher*. Cypher is a 'declarative, pattern-matching query language that makes graph database management systems understandable and workable for any database user, even the less technical ones.' [21].

Querying is definitely a strength of Neo4J. The join operations are designed to be extremely effective, because they are precalculated even before running the query.

4 Bibliographic Databases

This section provides a general overview of the state of today's bibliographic databases and describes different types of bibliographical databases. First, we take a look at publication databases and show some examples of how they compare to each other. Next, we describe databases storing bibliographic information about patent data and provide some concrete examples of these data sources.

4.1 Background

There is a growing need for services for finding descriptive records of relevant information sources. These services focus on collecting citation information and other related research data and attempts to make them searchable. Rise of the internet in recent decades has provided the opportunity to build an online, searchable literature databases that are accessible to anyone.

A "bibliographic database" is a database containing bibliographic records. A bibliographic record is an entry in a library catalog (bibliographic index) which contains fields necessary to identify a resource. These databases usually focus on a particular field of knowledge, and contain various types of bibliographic data (resources): books, magazines, newspapers, reports etc.

Bibliographic descriptions of patents is another big field of focus for several bibliographic databases. In this work, there will be a major focus on bibliographic databases, concentrating on patent data.

This chapter will focus on general descriptions of various, publicly available publication and patent databases from all over the world.

4.2 Publication databases

Databases containing bibliographic information about publications are called publication databases. The access to these databases is provided by an *academic search engine*. An academic search engine allows users to search for information related to various publication information. The distinction between an academic search engine and a simple publication database is not really clear, often it is used as a synonym those two terms are used interchangeably. The authors in [11] state that an academic search engine is a

'free web-based search service that incorporate added-value elements (citations, indicators and so on) which allow their use for research evaluation.'

This section provides a brief overview of the most popular academic search engines. It contains summarized information from [11].

4.2.1 Google Scholar

Google Scholar was first started in November 2004. It is considered to be the first complete search engine specializing in scientific literature. It is capable of extracting relevant information from various data sources, using many different crawlers and harvesters to search the web. As well as gathering information about scientific papers, it also harvests data about court opinions and patents.

One of the interesting features of Google Scholar is the ranking of the results which uses a form of its PageRank algorithm. The algorithm assumes that not every page is equal and assigns an importance score to each web page based on some calculated weight. In practice, this means that papers with the most citations and citations from prestigious sources are displayed first in the results.

Data Distribution

It is not easy to estimate how many papers are covered in Google Scholar, because Google is very restrictive in providing this kind of information to the public.

A survey computing the number of hits per year was carried out in 2013 by [11], however the source claims that it is a very broad approximation and may not be accurate enough. The figure 4.1 shows the distribution of data in Google Scholar by the type of documents and the table 4.1 shows the exact numbers for each document type. The presented data were created in a survey from 2013.

Academic papers	44,403,310
Citations	20,394,540
Patents	18,553,865
Books	11,467,605
Total	94,819,320

Table 4.1: The distribution of data by document type (Data from 2013).

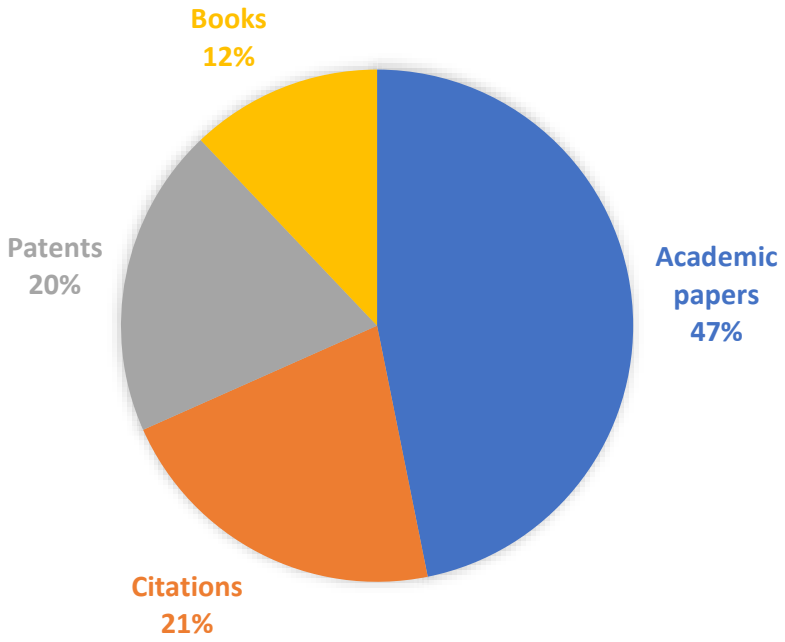


Figure 4.1: Distribution of data in Google Scholar by document type

From the Scholar's data distribution we can see that the majority of data consists of academic papers with almost 45 million records. They are followed by citations with more than 20 million records. The third common type are patents with 18.5 million records.

The recent estimates of researches conducted that Scholar could contain up to 389 million records [5].

API

Google Scholar is rather secretive when it comes to providing the indexed data and no API is available for developers to use. Therefore the retrieval

Publications	170 811 092
Authors	208 797 189
Affiliations	18 718
Journals	47 916
Conference series	4 025
Fields of study	195 957

Table 4.2: Data composition of Microsoft Academic Graph

of results is limited to using the web interface.

4.2.2 Microsoft Academic

Microsoft Academic Search is a search engine launched in 2009 by Microsoft Research Asia and was created to compete with Google Scholar [2]. It retired in 2016 and was replaced by Microsoft Academic.

Its main objective is to provide a platform which will be able to search through research papers, but also to provide tools for evaluation and analysis of science. Microsoft Academic puts emphasis on interaction with the user. The users can manage profiles and can suggest removals or corrections of existing data.

Data Distribution

A research conducted by [5] showed that Microsoft Academic contains about 170 million records, making it considerably smaller than Google Scholar with the estimate being around 389 million.

Microsoft Academic is powered by Microsoft Academic Graph which is a 'heterogeneous graph containing scientific publication records, citation relationships between those publications, as well as authors, institutions, journals, conferences, and fields of study.' [24].

The report from 2018 [30] shows the composition of data in Microsoft Academic Graph. The results of the report are displayed in the table 4.2.2.

API

Microsoft provides access to Microsoft Academic Graph data through *Academic Knowledge API*. It consists of four REST endpoints [31]:

- calchistogram – Returns a list of entities that satisfies some expression. Together with the expression, we can pass a list of attributes for which we want to generate a histogram.

- interpret – Returns a formatted interpretation of a user query using Academic Grammar.

  As an example, if we have a user query: *papers by john heynes after 2013*. An *interpret* endpoint will return a formatted representation using Academic Grammar as:

  And(Composite(AAȦuN=='john heynes'),Y>2013)

  This formatted query can now be used to call *evaluate* or *calchistogram* endpoints.

- evaluate – Returns a list of results to a query.

- similarity

4.2.3 Web of Science

Web of Science is a web-based scientific citation research platform. It gives access to multiple databases that reference cross-disciplinary research which allows for in-depth exploration of specialized sub-fields within an academic or scientific discipline [12]. WoS contains data about more than 20 thousands leading journals from 256 science disciplines. It also includes over 190 thousands conference proceedings and over 90 thousands books.

WoS makes data available from the following databases: [38].

- Current Contents Connect
- Data Citation Index
- Derwent Innovations Index
- KCI-Korean Journal Database
- MEDLINE
- Russian Science Citation Index
- SciELO Citation Index
- Zoological Record

4.3 Patent databases

Patent databases are a type of databases concerned with storing patent data. There are patent databases which contain full-texts of each patent along with images and other graphics. In this thesis, we are not interested in the full text of a patent nor images associated with them. There are also specific patent databases which collect only bibliographic information about patent applications as well as granted patents. There are some free, publicly available databases like USPTO and some which require to pay a subscription fee like PATSTAT database. There are also those which do not provide their data to public at all.

In this work, we will only focus on the sources of patent data which are willing to provide their data in some way.

4.3.1 What is a Patent?

A patent is a type of intellectual property which gives an exclusive right granted by the government for an invention. It permits the inventor to claim ownership of the invention and 'exclude others from making, using and selling the invention'[13]. A patent is always issued for a limited time.

4.3.2 Patent Types

The patents are classified into three main types: (The following definitions of patents are cited from [13].)

Utility Patents

> Covers inventions that function in a unique manner to produce a utilitarian result. Examples of utility inventions are Velcro hook-and-loop fasteners, new drugs, electronic circuits, software that is tied to some form of hardware, semiconductor manufacturing processes, new bacteria, newly discovered genes, new animals, plants, automatic transmissions

Design Patents

> A design patent (as opposed to a utility patent) covers the unique, ornamental, or visible shape or surface ornamentation of an article or object, even if only on a computer screen. Thus if a lamp, a building, a computer case, or a desk has a truly unique shape, its design can be design patented

Plant Patents

'A plant patent covers asexually reproducible plants (that is, through the use of grafts and cuttings), such as flowers (35 USC 161). Sexually reproducible plants.'

4.3.3 USPTO Patent Database

The United States Patent and Trademark Office (USPTO) is an agency of the Department of Commerce in the United States. It issues patents and registers trademarks to inventors and businesses.

It has its own searchable patent database. It is free to use and provides bibliographic data, full texts as well as images of patents issued in United States. The data is collected since 1790.

The searchable fields include:

- patent number
- issue date
- technical features in the patent
- keyword
- inventor's name
- company's name
- application date

4.3.4 Espacenet

European Patent Office (EPO) is the European patent database. It contains patent data on more than 100 million patent documents from 80 different countries and 100 patent authorities [29].

There are two main databases in Espacenet:

EP Database

Contains the collection of patent applications published in Europe by the European Patent Office.

WIPO Database

World Intellectual Property Organization (WIPO) is an organization gathering global intellectual property services.

4.3.5 CIPO

The web site of the government of Canada [32] describes the Canadian Intellectual Property Office (CIPO) as 'a part of Innovation, Science and Economic Development Canada. CIPO is a Special Operating Agency (SOA) and is responsible for the administration and processing of the greater part of intellectual property (IP) in Canada.'

Searchable fields include:

- keyword
- patent document number
- patent date

4.3.6 Google Patents

The Google Patents is a web platform which is able to search patents from around the world. 'Google Patents includes over 120 million patent publications from 100+ patent offices around the world, as well as many more technical documents and books indexed in Google Scholar and Google Books, and documents from the Prior Art Archive' [35].

4.3.7 PATSTAT

PATSTAT is a worldwide patents database created by European Patent Office. It contains data on more than 67 million patent applications and 35 million granted patents from many countries [7].

5 Data Gathering

The main part of the work done in this thesis is the process of gathering large amount of data which will be made accessible in some way. The acquired data should focus on gathering data describing various patents.

This chapter describes the taken approach, what technologies were chosen and employed and the reason for the choices. It later showed that those choices also had some disadvantages and limitations, so they are also listed and described. Then we list sources of data that were identified and acquired and the process of the data acquisition is given. Next, we describe the process of preparing the data and the process of preprocessing it for insertion to the main database. Later section will describe how the data is being stored and how indexes were used to improve the performance of querying.

5.1 Selected Approach

In order to store the acquired data, appropriate database had to be chosen. The sources of data come in different shapes and structures. We needed a fast and secure storage that would be able to store semi-structured data containing tens or hundreds of millions of records. Due to the limitation of the relational database model (as listed in 2.2), the decision was to look for solutions in a non-relational approach.

After the database model was decided, it was necessary to chose one from various non-relational (NoSQL) databases and pick the one that would best suit our needs. The comparison given in 3 gives an idea of the available options.

5.1.1 Database Requirements

The requirements for the database were mainly the ability to store large amounts of data. Another important aspect was the performance of the database. Because the data would first have to be inserted in the database, good insertion performance would be essential. It was not excluded that in the future the database will be accessed in concurrent manner, with many users querying the database at the same time. So the concurrency of the database also had to be considered. Another thing was querying performance. It was necessary for the database to be able to search through the stored data and respond to queries in a reasonable time. Last requirement

on the database was security. The database would have to provide some security measures to prevent from outside manipulation.

In the end, the decision was between using MongoDB, ElasticSearch or Solr.

As many public databases can be acquired as a bulk file containing a set of documents, this was an indicator that it would be appropriate to chose a document-based NoSQL database. After careful considerations, it was decided that the most appropriate database for our needs would be MongoDB.

5.1.2 Technology

Java programming language was used for the applications developed in this work. One of the reasons for choosing Java was that it is a platform-independent language. MongoDB provides a support for Java thanks to its Java API driver. MongoDB's aggregation pipeline, where operations in the pipeline can be chained after each other, is very similar to Java Stream API.

5.1.3 Limitations of MongoDB

MongoDB version 4.0.5 was chosen, because it is fast, secure and provides schema-less data storage model. This provides an advantage when integrating multiple sources of data with different structures.

Because it was my first time working with NoSQL, my choice of using MongoDB as a primary database was mainly based on theoretical information and not on experience. This resulted in some inconveniences and limitations surfacing after some time later in the development stage.

Full-text search

One of these inconveniences was the full-text searching capabilities of MongoDB. From version 3, MongoDB introduced the *text* index. Text indexes provide the ability to search on string content [28]. Unfortunately, there can only be one text index on the entire collection, although a it can be a 'compound index' which can textually index multiple fields. The result of that is the fact that it is not possible to limit search only to specific fields, because the text index is constructed on the collection level. Therefore it is not easy to implement searching on specific fields and there needs to be some workaround in place which is further described in section 5.6.2.

Another limitation of MongoDB's full-text search is the inability to use complex boolean operators in queries, like in Lucene. MongoDB full-text query are by default evaluated with OR operators in between tokens or we can specify the exact term to search for.

5.2 Data Sources

Appropriate data sources had to be identified and studied. There are several major databases that provide the data for download. Some provide it for free, others are licensed and require a subscription fee.

5.2.1 Data Structure

Each data source differs in the structure it comes in. Some are in XML format, some of them are in JSON format and others are in CSV format, ready to be loaded to an SQL database.

Because of this diversity in the structure of each source, a method to unify the structure of multiple data sources had to be created.

The descriptions of each identified data source are given in the following sections. The format, data volume and the structure of each data source is discussed.

5.2.2 USPTO Data

The Unites States Patent and Trademark Office provide static bulk data of patents issued in the United States. The data is provided in a set of files available free for download.

Format

The format of the available data is XML. The USPTO collects bibliographic data about patent grants issued weekly since the year 1976. Only a subset of this data was collected. In 2005, the USPTO changed the format of the provided data to XML.

Structure

Each patent record contains large number of fields and the structure is very complex with many nested fields of different types. A simplified example of one of the records from USPTO can be seen in the Appendixes section 10. Unfortunately it later showed that the structure did not stay the same and

some fields were renamed throughout the years. This made data unification process harder. It will be discussed later in the section 5.5.

Some of the most commonly used fields are described in the table 5.1.

Field Name	Field Type	Description
id	string	ID of the USPTO record
title	string	Title of the patent
applicants	array of strings	Applicants for the grant
inventors	array of strings	Inventors of the grant
year	int	Filing year
abstract	string	Abstract

Table 5.1: Structure of USPTO data.

Volume

The size of data acquired from USPTO is 107 GB in its extracted form in XML. The number of patent records contained in the data set is roughly about 3.6 million records. It contains data since the year 2000. Unfortunately, until the year 2005, different format for storing the data was used, so data before the year 2005 were scrapped and only data from that year stayed.

Data Download

The data can be downloaded from their official website[1].

5.2.3 Microsoft Academic Graph Data

Data from Microsoft Academic Graph were obtained using the Open Academic Graph project [17]. It is released under the ODC-BY license. Data is provided in a set of downloadable bulk files containing publication records from all over the world [20], [17].

Format

The data from Microsoft Academic Graph is stored in a JSON format with each publication record separated by a new line.

[1]The URL addresses are visible in the Appendix section 10

Structure

The most commonly used fields in the data and their description are listed in the table 5.2.

Field Name	Field Type	Description
id	string	Microsoft Academic Graph ID
title	string	Title of the publication
authors.name	string	Author's name
year	int	Published year
keywords	array of strings	Keywords for the publication
publisher	string	Publisher
isbn	string	ISBN
abstract	string	Abstract

Table 5.2: Structure of Microsoft Academic Graph data obtained from Open Academic project.

Volume

The amount of data download from the Open Academic project is the largest data source identified. It contains bibliographic information about 166 millions of scientific papers. No full-texts or images are part of the data set.

Data Download

Data from the Open Academic Graph project can be downloaded from the official website of the project.

5.2.4 PATSTAT Data

Format

The format of PATSTAT data is CSV, separated into files and ready to be loaded to an SQL database.

In order to convert the data into the required format, the process of converting the data from CSV to JSON was quite complicated and will be further discussed in later sections.

Structure

The following table 5.3 shows the native structure of the PATSTAT data, when converted to JSON format.

Field Name	Field Type	Description
appln_id	integer	Application ID
title.title	string	Title of the patent
appln_filing_date	string	Date of filing the patent application
authors.party.name	string	Name of the author

Table 5.3: Structure of PATSTAT patent data.

Volume

The compressed size of the raw data is around 10 GB and once the data is loaded into the database, it rises up to around 100 GB.

Data Download

The PATSTAT database is available for purchase on their official site and offers multiple subscription plans containing various data sets.

There are two product lines for PATSTAT data. The PATSTAT Global containing bibliographic data with 'more than 100 millions of records' and the PATSTAT EP Register containing 'legal status data on published European and Euro-PCT patent applications.' [37]

5.2.5 DBLP Data

DBLP provides their data as one big file containing bibliographic records.

Data are available for download from their official site.

Format

The format used for DBLP data is XML. It contains a long list of XML records. The root element has several million child elements, but usually no element is deeper than level three. The Figure 5.1 shows the common structure in the DBLP data.

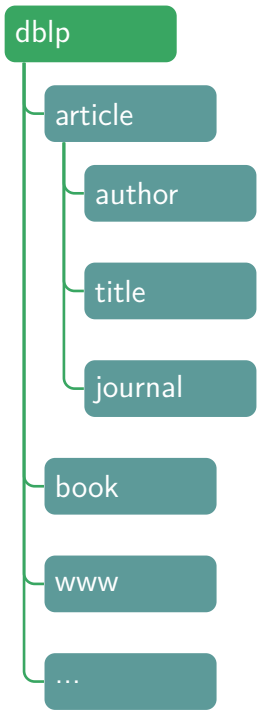


Figure 5.1: A common structure of DBLP's XML file

Structure

The record in the DBLP data contains the following fields:

Field Name	Field Type	Description
author	string	Author's name
title	string	Title of the paper
year	integer	Year of publication
pages	integer	Number of pages
publisher	string	Name of the publisher
school	string	Name of the school
isbn	string	ISBN of the paper

Table 5.4: Structure of DBLP data.

Publication records are inspired by the BibTex syntax and can be one of the following elements. The following list is extracted from [33]:

- article – An article from a journal or magazine.
- inproceedings – A paper in a conference or workshop proceedings.
- proceedings – The proceedings volume of a conference or workshop.

- book – An authored monograph or an edited collection of articles.
- incollection – A part or chapter in a monograph.
- phdthesis – A PhD thesis.
- mastersthesis – A Master's thesis. There are only very few Master's theses in dblp.

A sample excerpt from DBLP can be seen in the appendix 10.

Volume

The DBLP database stores around 4.5 million records as of 2019 [34]. Figure 5.2 shows the total numbers for papers of different publication types.

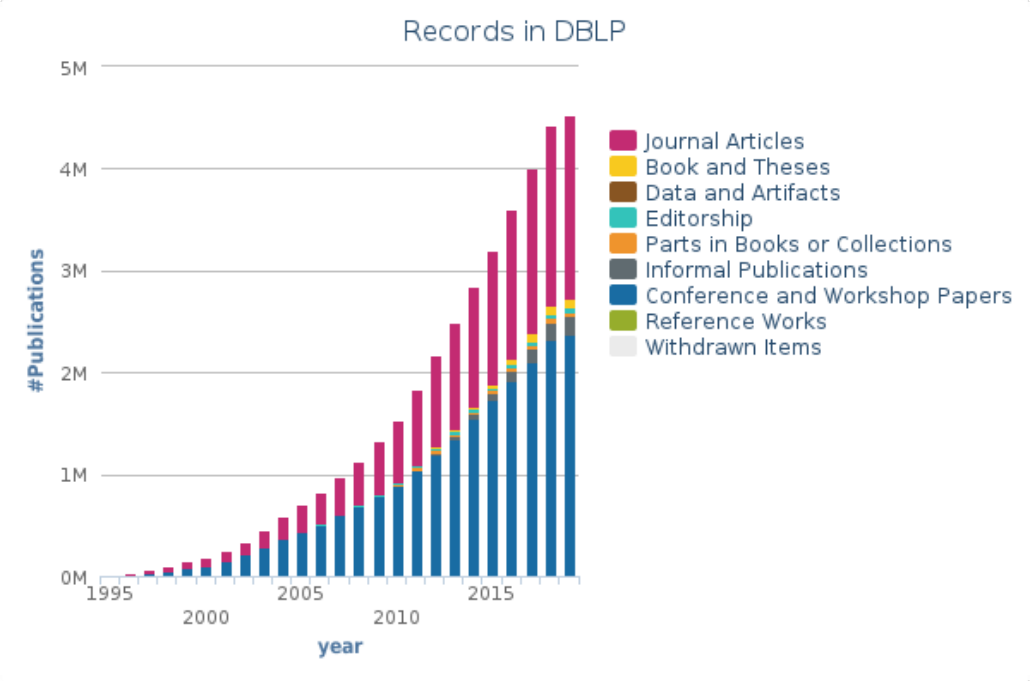


Figure 5.2: Statistics of data in DBLP

5.2.6 Summary

The listed data sources were identified as good candidates to be included in the main database. Unfortunately, not all data sources were prepare dto be inserted to the main database. The Springer LOD data source was not prepared in time, so it is not included in the main database. The data from

Czech Trademark Office were acquired, but was not suitable for insertion to the database.

Before the data sets could be inserted to the main database, it was necessary to do some preprocessing of the data. The activities of preprocessing and inserting the data to the main database will be described in the following sections.

Table 5.5 shows the summary of all the data sources that were acquired with some additional information about them.

Data Source	No. of Records	Data Range	Extracted Size (GB)
USPTO	3.6M	2005 – 2018	104
M.A.G.	166M	1998 – 2016	103
DBLP	4.5M	1995 – 2018	2.35
PATSTAT	> 100M	2005 – 2018	100
Total	274.1M		309.35

Table 5.5: Table summarizing basic information about all the acquired data sources in their native form.

5.3 Data Preprocessing

Data we obtain from public databases come in various formats, each having different structure. For example, data from USPTO come in a set of files in an XML format. The PATSTAT data come in a CSV format logically separated into several files representing tables in a Relational database. Obviously, these two data sources cannot be left in its original form, because we need some unified fields that we want to access across all the data sources. This means that they cannot be immediately inserted into the database. Apart from different formats, data sources come with varying structures. That means that each source can have different names for the same fields. For example, an author of some publication or a patent in one source is called an *inventor* and in the other one it is a nested field called *party*. The following snippets from two different data sources illustrate the issue:

```
...
<parties>
 <applicants>
  <applicant>
   <addressbook>
    <first-name>Matthew</first-name>
    <last-name>Kosh</last-name>
    <address>
     <city>Seattle</city>
     <state>WA</state>
     <country>US</country>
    </address>
   </addressbook>
  </applicant>
 </applicants>
</parties>
...
```

This is a snippet from USPTO data set, we can see that the name of the applicant is nested deep inside the structure, so in order to extract it, we need to traverse the tree from the root [2]. In this case, for extracting the first name of the applicant, the path would look like:

parties.applicants.applicant.addressbook.first-name.

The structure of the record from Microsoft Academic Graph looks much simpler with the path to the full name being: *authors.name*. The listing below shows the structure in JSON format.

```
...
"authors": [
    {
      "name": "jiawei han",
      "org": "department of computer science university of
         illinois at urbana champaign"
    },
    {
      "name": "micheline kamber",
      "org": "department of computer science university of
         illinois at urbana champaign"
    }
]
...
```

[2] In XML, it is possible to access nodes directly using relative path in xPath, but we will be working with JSON format, where this is not easily possible

As we can see from the two previous listings, each source can have different structure and data may be stored in varying formats. That is a problem for multiple reasons. First one being that in order for the data to be readable in a unified manner, we would have to create separate code for extracting information for every data source. Another problem is that indexes in MongoDB are created on a set of fields specified by their field names. But in this case, every data source has different field names pointing to the same information.

Because of these reasons, it is not beneficial to store the data in their native structure and it is necessary to create a unified structure that all the data sources would share. Also, it was necessary to differentiate between publications and patents, because each type of data has its own unique fields. The listing 5.3 illustrates the unified structure which was used for patent data sources in JSON format.

```
...
{
   "title":"string",
   "abstract":"string",
   "year":"integer",
   "number":"string",
   "authors":[
      {
         "name":"string"
      }
   ],
   "owners":[
      {
         "name":"string"
      }
   ],
   "keywords":[
      "string",
      "string"
   ]
}
...
```

Figure 5.3: Unified structure for patent data sources in JSON format

Every data source that is inserted into the sources database, needs to have this structure in the top level of each document. Figure 5.4 shows the phases of preparing the data until insertion into the database.

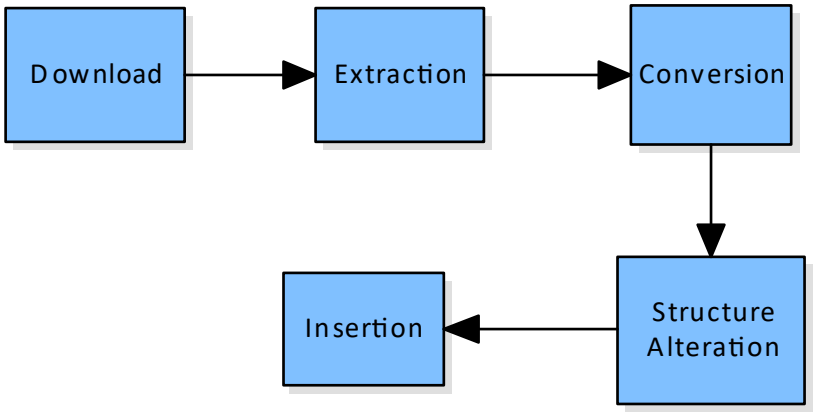


Figure 5.4: Data preprocessing pipeline

Data Download

The process of preparing the data starts with downloading it from the source website. As different databases are downloaded in different ways, this process cannot be easily automated and therefore it needs to be done manually.

Data Extraction

Often, data comes in a compressed form and needs to be extracted. A tool for bulk extraction was created as a part of the software solution.

5.4 Data Conversion

The next step is the conversion of data. As described in section 3.1, MongoDB accepts only JSON format, so every data source first needs to be converted to that format.

Depending on the source data format, two approaches were used while working on this thesis:

5.4.1 XML Conversion

Data coming in an XML format are quite straight forward to convert. A utility tool was created, able to convert an XML file to JSON format and saves it to an output directory. It is also possible to provide a path to a directory and convert all XML files in bulk. If the *includeSubdirectories*

flag is set, all the XML files from sub-directories are processed too. The outputted JSON files are saved in the target directory while maintaining the original folder structure of the directories and sub-directories. The ability to convert in bulk came in handy as it was necessary to convert hundreds of files at once. The conversion to JSON format was done using a Jackson library for Java.

5.4.2 CSV Conversion

Data from PATSTAT are delivered as multiple CSV files corresponding to the tables in a relational database. It was not possible to create the desired JSON structure from these CSV files, so an alternate approach had to be used.

The process used to create the JSON was the following: First, relational database was created using the set of CSV files. Once all the data were inserted in the database, a query was constructed to extract the data in the required JSON format. SQL version 12.2 added a function called `JSON_OBJECT` which returns a JSON object from the specified query.

5.5 Column Mapping

Before the process of inserting the data into MongoDB, we needed to alter the structure of the data into the unified structure shown in Listing 5.3.

For this, a special file was designed which was used to map unified fields to paths pointing to those unified fields in the original data.

The structure was altered using column mapping. On the following code listing on Figure 5.5, we can see an example of a mapping file used to associate unified fields to their corresponding paths in the original file.

Each field in the mapping, like `title` on line 3, is assigned a path pointing to the field in that specific source of data. For example, the listing above shows the mapping file for the USPTO data (see page 83).

In case of mapping an array, a path to the parent element of the field we want to map is present, indicated by the name `array-root`, see line 11. The key named `values` then points to the actual value for that element. If each element of the array contains multiple fields, we can merge them together by specifying both of them in the mapping. See line 12 in the listing above.

We can also specify multiple paths to the same field. This is especially useful, when the data source changes structure over time which was not uncommon to happen.

```
{
   "uspto":{
      "title":"/us-bibliographic-data-grant/invention-title/
         content",
      "abstract":"/abstract/p/content",
      "year":{
         "path":"/date-publ",
         "format":"YYYYMMDD"
      },
      "authors":[
         {
            "array-root":"/us-bibliographic-data-grant/us-
               parties/inventors/inventor",
            "values":[
               "/addressbook/first-name",
               "/addressbook/last-name"
            ]
         }
      ],
      "owners":[
         {
            "array-root":"/us-bibliographic-data-grant/us-
               parties/us-applicants/us-applicant",
            "values":[
               "/addressbook/orgname"
            ]
         }
      ]
   }
}
```

Figure 5.5: A sample mapping file

The process of how the mapping was used will be shown in a later section 7 describing the process from an implementation perspective.

5.6 Inserting to the Database

Finally at this point, the data should be in a JSON format having correct structure and ready to be inserted into MongoDB.

The database will hold two major types of data. First type is data from publication sources holding papers like books, journal, articles, magazines, newspapers, reports etc. The sources of data containing this type of inform-

ation are Microsoft Academic, DBLP, Springer LOD. Another type of data will come from patent data sources. These include USPTO, PATSTAT.

There were two options to consider. Either create a separate MongoDB collection for each data source. The second option is to use only one collections for each type of data. Each option has its advantages but also drawbacks.

The advantages of the first approach, having separate collection for each data source has the following advantages:

- Smaller size of indexes – This would be beneficial, if we often performed searches only on one data source. It showed it is not the case and we usually want to search from all the data sources.
- Readability – Having each data source in a separate collection is definitely more readable.

The drawbacks of the first approach are:

- Large number of indexes – Each data source would have to contain its own set of indexes for its fields. This would be hard to maintain when the number of data sources grows larger.
- Many queries – If we want to search in every data source, we would have to run the query multiple times. It showed that we will be searching through all data sources very often.

The second approach, having only two global collections, has the following advantages:

- Easier DB inserts – Everything goes to the same collection.
- Few indexes – As there are only two collections, there will only be two sets of indexes, one for each collection.
- Single query – Query will be run only once on the global collection.

The drawbacks of the second approach are:

- Large size of collection – The collection can grow quite large.
- Harder to search only one data source – To search a single data source, we need to add an origin source id to every document.
- Less readable

It was decided that the second approach is more appropriate for our needs, so only two global collections were created, each holding one type of data. They were named *publication* for papers and *patent* for patent data respectively. The process of inserting the JSON documents into the database programmatically will be described in later section 7.4.6 using streaming API.

5.6.1 Indexing

Proper use of indexes is essential for higher performance in MongoDB.

Several indexes were used to increase the performance of queries. As described in 3.1, MongoDB uses two types of indexes. Regular indexes and text indexes. The biggest difference between these two is that a text index can be used for full-text searching. Without these indexes, the whole collection would have to be scanned for every query which would mean a decrease in performance.

In contrast to ElasticSearch, collections in MongoDB do not have an index on its every single field created by default, but have to be created manually and only specified fields are part of the index. It was necessary for the index to contain those fields, on which we will perform searches. Each collection contains a regular single field indexes on fields which do not need the support of full-text searching capabilities. Fields, for which a single field index was created are:

- year – Year of publication of a patent or a publication
- keywords – The list of keywords associated with the record
- number – The number of a patent

Fields that will need to be searched using full-text search and thus have a text index present are:

- title – The title of the patent or a publication
- abstract – The abstract of a patent or a publication
- authors – The list of authors associated with the record
- owners – The list of owners of a patent. Does not exist for publications.

The table 5.6 shows which fields should be present in patents and publications data sources.

Concrete examples of measurements of the performance while searching with and without indexes can be seen in the last chapter 9.

Field name	patents	publications
title	x	x
abstract	x	x
authors	x	x
year	x	x
keywords	x	x
number	x	
owners	x	

Table 5.6: Existence of fields in patents and publications

5.6.2 Searching

As described in the section 5.1.3, it is possible in MongoDB to create a text index on multiple fields. However, when searching, it is not possible to specify fields in which the search should be performed. This issue was discovered later in the development stage. Therefore it was necessary to create some workaround, to be able to search only in specific fields. The possibility to use full-text search on specified fields was implemented using a combination of a regular index and a regular expression search.

The search on the database works in two modes: Selective and non-selective search. In selective mode, MongoDB will search only in a single field, specified by the user. The search uses a combination of a regular index and a regular expression matching. This enables to limit the search on a single field. The drawback of this approach is primarily performance, because regular expression matching is much slower than using a text index. The difference in performance when using a text index is discussed in the chapter 9. In non-selective mode, text index is used to perform a full-text search on all fields that are included in the text index. The performance of this method showed to be much faster than using a regular expression.

6 Software Solution

The final solution should create a platform which will be able to search for IPR-related data. This data mainly includes bibliographical information about patents and publications. This work focuses on creation of the large database concentrating data from various data sources and integrating them together. Additionally, it allows for an access to the database using an API which enables users to query it. To demonstrate the functioning of the API, a demonstration web application should be built. It should be a simple web application that can call the API and display the returned results. Some basic visualization of the returned data should also be created. Lastly, an application for high level data administration should be developed.

The following chapter will present a global view of the whole system. It will give an explanation of the individual components and classes. It will also describe the provided REST API and locations of external metadata files about data sources.

The Figure 6.1 shows the component diagram, showing the individual components of the system and relationships between them.

The solution consists of 5 main components:

- Sources DB
- Data Sources DB
- Data acquisition application
- REST API application
- Web interface

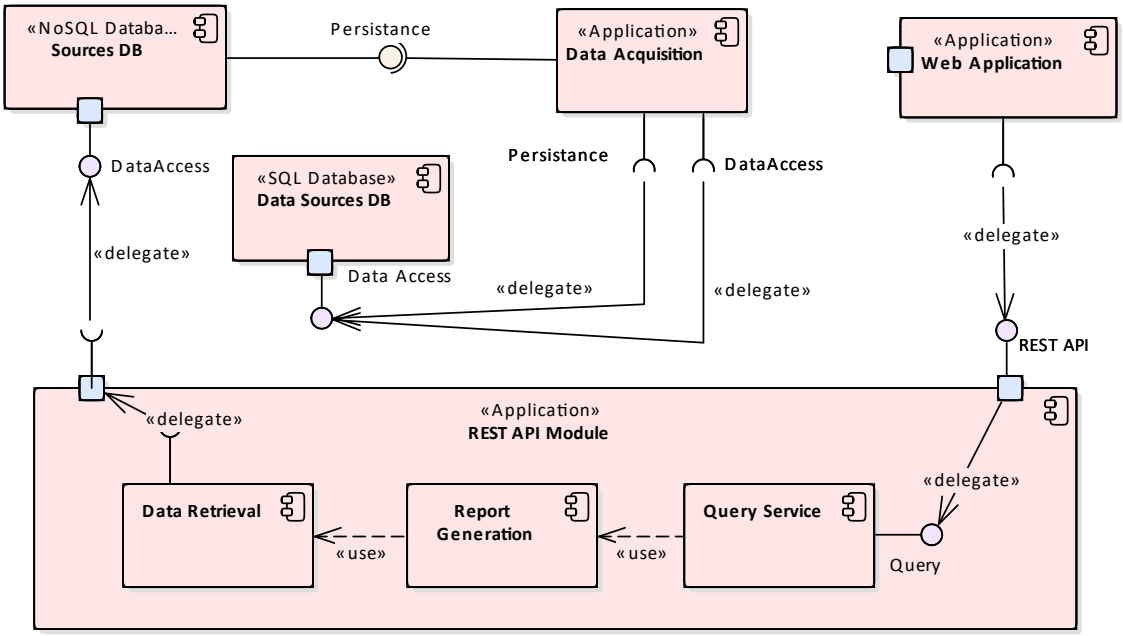


Figure 6.1: A component diagram of the system

6.1 Sources DB

The *sources* database is the main database containing the collected data from various data sources. The data is stored in a document-based NoSQL database MongoDB. The process of gathering the data and filling the database was the content of the previous chapter 5.

6.2 Data Sources DB

The Data Sources database is a relational database running on *MariaDB*. It is a complementary database to the sources database and contains metadata about the acquired data sources. On top of that, it is supposed to speed up the performance of duplicate queries by storing previously run queries along with the returned results.

The Figure 6.2 shows the entity relationship diagram of the data sources database.

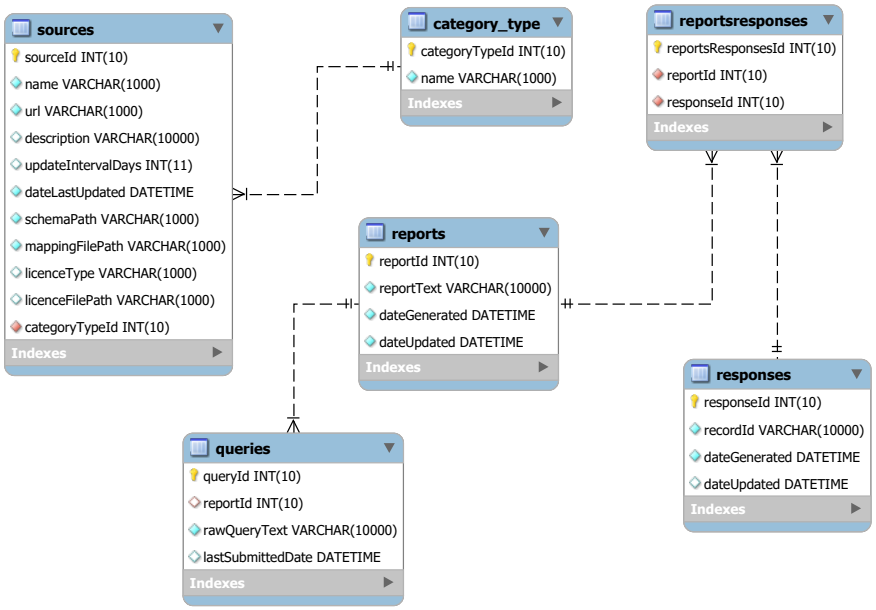


Figure 6.2: ER diagram of the data sources database

6.3 Data acquisition

The data acquisition application is meant to be used primarily by data administrator/developer to simplify and streamline the process of managing meta data about data sources and loading new data to the Sources database. It also provides some basic tools needed for data preprocessing like bulk extraction of ZIP files or bulk conversion of XML files to JSON format.

6.4 REST API application

The REST API application's main purpose is to provide an interface which will be used by clients. The interface should facilitate access to the Sources database. The client will be able to call the API with a query which will be run on the Sources database and retrieved results will be returned back to the client as a JSON response.

6.5 Web interface

The web interface is not the main focus of this work so only a simple demonstration web application will be created to demonstrate the functioning of the REST API and simple visualization of returned results.

7 Data Acquisition Application

The data administration application was created mainly to simplify tasks that are necessary to create and manage data sources. The application is intended to be used by an administrator/developer.

It's main use cases are:

- Loading data to the sources database
- Creating metadata for new data sources
- Configuring file locations of files related to data sources – mapping files, structure files
- Managing the metadata for data sources – creation, deletion, update
- Managing expiration of data sources

The application is written in Java programming language with support of JavaFx. JavaFx is a software framework used to create mainly desktop applications in Java. It was first released in 2008 and was intended to replace Java Swing for creating graphical user interface.

List of all the used libraries in the application can be seen below on table 7.1.

Name	**Version**
MariaDB JDBC	2.4.0
MongoDB Java Driver	3.9.1
Jettison	1.4.0
Apache Commons Lang	3.0
JSON	20180813
Apache Commons DbUtils	1.7
Jackson Core	2.9.8
TestFX	4.0.1
JUnit	4.12
Guava	14.0.1

Table 7.1: Used libraries in the application

This chapter will describe the application's architecture, it's main packages and classes within them and also the relationships between the classes. Then the process of data loading will be described. The last section will outline the testing of the application's main functions.

7.1 Architecture

The architecture follows an Model-view-controller (MVC) architectural pattern which separates the application into three logical components: Data presentation, data logic and data model.

The Figure 7.1 shows the package diagram of the application.

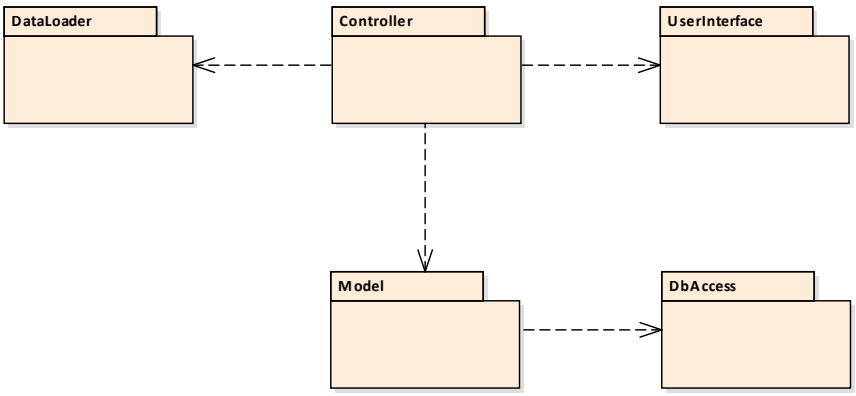


Figure 7.1: A package diagram of the data administration application

The application consists of 5 main packages. The controller package is sitting in the middle and is used to communicate requests from the presentation component to the data model. The presentation component consists of the layout of the user interface and the bindings to the controller's fields. The `dataLoader` package is also used by the controller to execute tasks which serve the purpose of loading data into the main source database. The data model is being called from the controller to execute domain specific tasks. It also contains domain classes and provides access to the storage unit, in this case, the database.

The following sections will now describe each package in more detail.

7.2 UserInterface package

`UserInterface` package contains files defining the user interface of the application. JavaFX uses *FXML* files to describe the layout of the user interface. FXML is a XML-based language which provides a structure for building user interfaces. The main advantage of using FXML files is that it provides a separation of the application logic from the interface.

The application's graphical interface consists of a single window, logically separated into multiple FXML files. The Figure 7.2 shows the graphical user interface and how it is separated into FXML files.

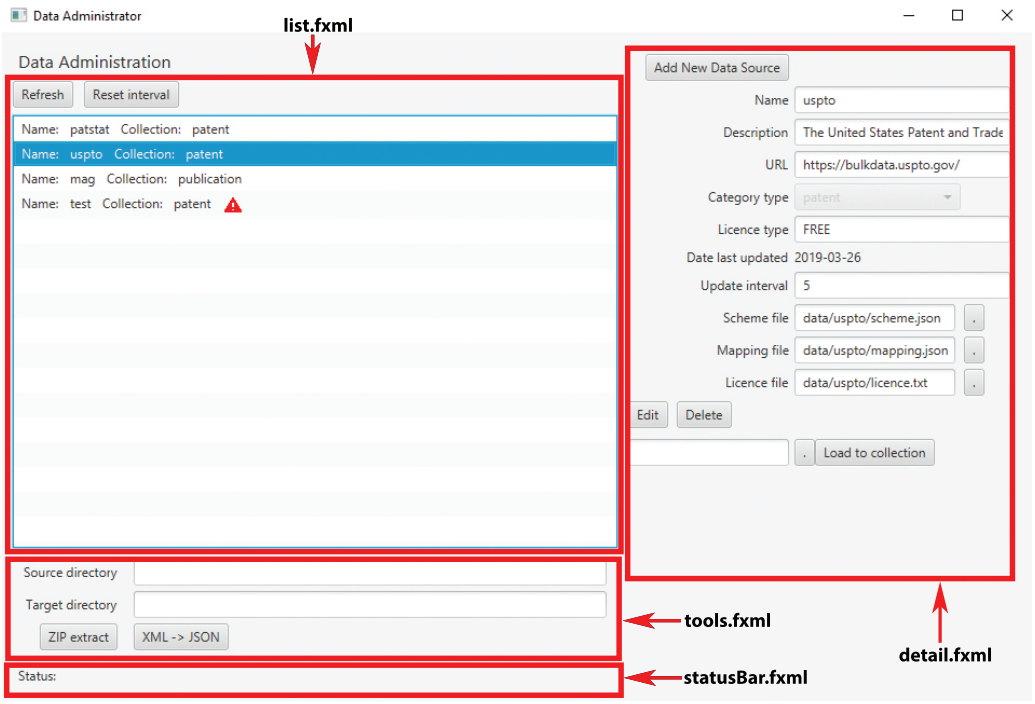


Figure 7.2: A GUI of the data administration application

There are 5 FXML files describing the UI:

- *list.fxml* – Displays a list of items, where each item shows the name of the data source and the collection. Each item can be selected.
- *detail.fxml* – Based on the item selected in the list, the appropriate window is rendered in the details panel.
- *tools.fxml* – Provides access to some tools useful before loading the data to the main database.

- *statusBar.fxml* – Displays information about currently running tasks.

7.3 Controller package

For each FXML file, there is one controller class. A controller is bound to the UI components and is responsible for initializing them. It is instantiated by the FXML loader.

The controller package contains the following classes:

- *ListController* – The list controller class contains a *ListView* control with a list of data sources. It is populated with records from the data sources database's *sources* table (see 6.2).

  It contains a custom ListView cell implementation which allows for custom formatting of the list items. Each item displays a name of the data source and a collection in which it is stored.

- *DetailController* – The detail controller contains a form with the metadata for the list item selected in the ListView and dynamically changes based on the selected list view item. It also initiates tasks for loading the data to the database. These tasks are run on a separate thread. In case of the loading process throwing an exception, it is caught by the task and a message is displayed to the user.

- *ToolsController* – Tools controller enables running some basic file handling operations, like extracting a directory containing zip files or converting XML files to JSON files. These tools provide the benefit that they are able to handle bulk operations. They also maintain the original directory structure after the operation.

- *StatusBarController* – Controller for displaying information about running tasks and also displays results of some operations.

- *ErrorController* – Error controller is bound to a separate window and displays any run-time exceptions that occurred in the application. The whole stack trace is displayed.

7.4 DataLoader package

The data loader package contains all the logic for loading data sources to the main sources database and is the most important part of the data administration application. It is also a key part of the whole thesis which is concerned mainly around creating a large database of IPR-related data.

The figure 7.3 shows the class diagram of the data loader package.

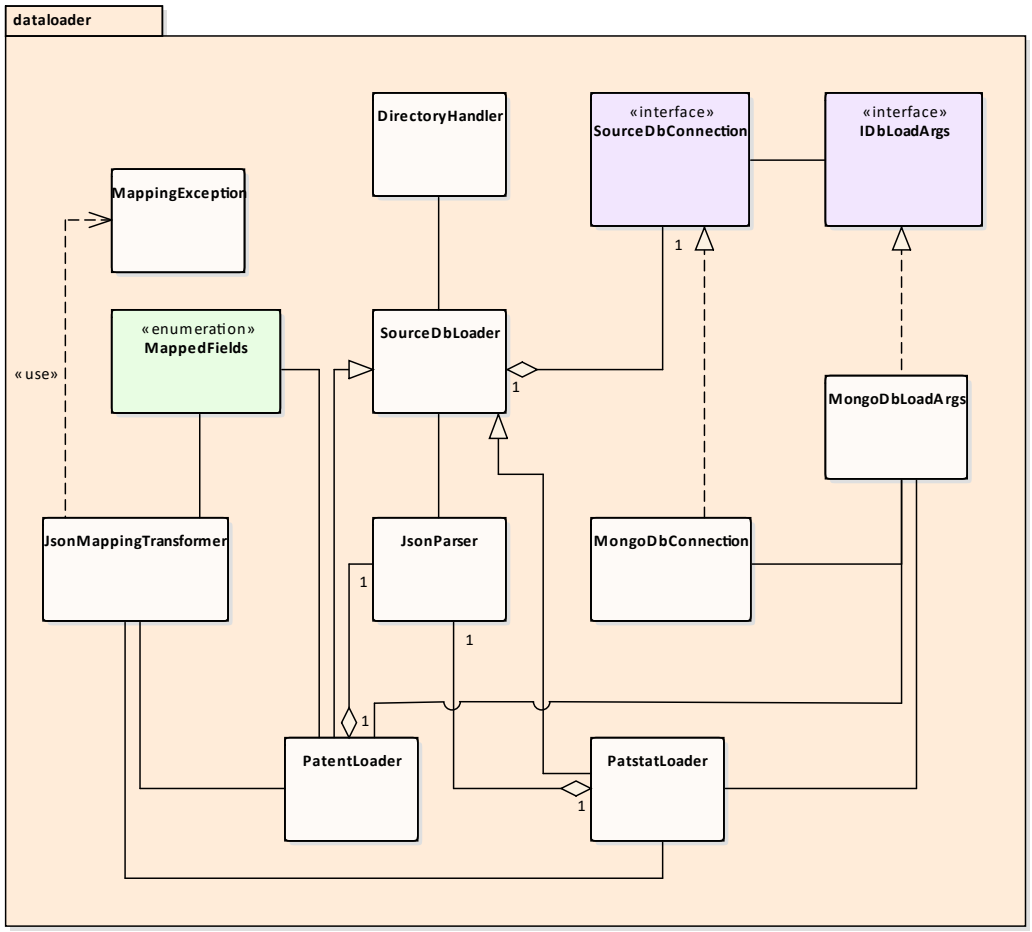


Figure 7.3: A class diagram of the data loader package

7.4.1 SourceDbConnection

This interface specifies methods for connecting to the sources database. Concrete implementations of database connections will implement this interface to connect to the target sources database and to insert data. At the time of writing this thesis, MongoDB NoSQL database was used. If there is a need to change the database, there will be another concrete class implementing this interface.

- `void connect();`
- `void insert(IDbLoadArgs args);`

- `void disconnect();`

7.4.2 MongoDbConnection

The concrete class implementing the `SourceDbConnection` interface. It provides access to the MongoDb database and is able to insert data into collections.

7.4.3 SourceDbLoader

`SourceDbLoader` is an abstract class for loading the data to the sources database.

It has three methods with two of them being abstract. The signatures of the methods are the following:

```
void loadFromDirectory(String dirPath,
                String[] extensions)
      throws IOException, MappingException
```

Method `loadFromDirectory` loads all the files with the specified extension from the directory (including its subdirectories) to the target database.

```
abstract void insertFromFile(File file)
           throws IOException, MappingException
```

`insertFromFile` method inserts a list of documents from a file to the target database.

```
abstract void preprocessNode(JsonNode nodeToPreprocess)
                throws MappingException, IOException
```

Does the preprocessing of a json node, so that it follows the defined unified structure described in 5.3.

7.4.4 PatentLoader

The `PatentLoader` class extends the `SourceDbLoader` class. It is responsible for loading data from the USPTO data source (see 5.2.2). The most interesting part of this class is the `preprocessNode` method.

It uses the `JsonMappingTransformer` class to manipulate the JSON document, so that it contains the unified structure as described in Figure 5.3. This is done using the provided mapping file which specifies the paths to each field in the data source. The path for each field is read from the mapping

and extracted from the current document at that path. The value found is then moved to the top level of the document.

As an example, in USPTO, the title field is located in the following path: */us-bibliographic-data-grant/invention-title/content*. This path is specified in the mapping file for the USPTO data source. The mapping also specifies that the target field should be named *title*. So during preprocessing of the document, the algorithm looks at the value at that path, moves it to the top level and names the field *title*.

In the code, this can be done using the `putValueFromPath` method of the `JsonMappingTransformer` class:

```
1 JsonMappingTransformer.putValueFromPath(mappingRoot,
    MappedFields.TITLE, nodeToPreprocess);
```

It is also possible to map arrays in a similar way. An example is the *authors* field which is an array, because a record can have multiple authors.

```
1 List<String> authorsList = JsonMappingTransformer.
    getValuesListFromMappingArray(mappingRoot,
2             MappedFields.AUTHORS,
3             nodeToPreprocess);
4 JsonMappingTransformer.putArrayToNode(authorsList,
5                 nodeToPreprocess,
6                 MappedFields.AUTHORS, "name");
```

Another thing to note is that because the structure of the data changes over time, there is a necessity to handle multiple paths to the same field. An example is again the USPTO data source. Until 2013, the path to the authors field was: */us-bibliographic-data-grant/us-parties/us-applicants/us-applicant*, but since then, it changed to */us-bibliographic-data-grant/us-parties/us-applicants/us-applicant*. Therefore in the mapping, it is possible to specify multiple paths pointing to the same field. For the authors field, the mapping file would look like the one on Figure 7.4:

In the code, all the paths to the same field will be processed one by one and once we find some value for the particular option, it will take the value for that option and stop iterating over the others.

7.4.5 JsonMappingTransformer

This class handles manipulation of the JSON files using the provided mapping configuration. It loads the mapping file and transforms a JSON node according to the configuration specified in the mapping file.

```
"owners": [
  {
    "array-root": "/us-bibliographic-data-grant/us-parties/us-applicants/us-applicant",
    "values": [
      "/addressbook/orgname"
    ]},
    {
    "array-root": "/us-bibliographic-data-grant/us-parties/us-applicants/us-applicant",
    "values": [
      "/addressbook/first-name",
      "/addressbook/last-name"
    ]}
]
```

Figure 7.4: Specifying multiple paths in a mapping file

7.4.6 JsonParser

`JsonParser` class reads an external JSON file, parses it, calls the preprocessing methods and creates a final list of documents which are ready to be inserted into the target database.

The process of parsing the JSON is done using the *Jackson* library. Jackson provides a streaming API which enables to create very fast JSON parser, but the disadvantage of this approach is that it can be a little more difficult to use, because everything in the JSON data has to be handled in the code.

The code listing on figure 7.5 shows the process of parsing the USPTO data to a list of documents. The *Document* class is a MongoDB's representation of a document.

```java
/**
* Streams the file and from its contents creates a list of documents to be added to the database.
* @param file - File to be parsed
* @param loader - The callback loader to call for node preprocessing
* @param arrayName - Name of the array element to search
* @return - List of parsed documents to be added to the database. If there is an error parsing
* the file, an empty list is returned
*/
public List<Document> parseFileStreaming(File file,
            SourceDbLoader loader,
            String arrayName) {
List<Document> docs = new ArrayList<>();

JsonParser parser = new MappingJsonFactory()
        .createParser(file);
JsonToken current = parser.nextToken();
// read a document at a time
while (parser.nextToken() != JsonToken.END_OBJECT) {
    String fieldName = parser.getCurrentName();
    current = parser.nextToken();
    if (fieldName.equals(arrayName)) {
        if (current == JsonToken.START_ARRAY) {
            while (parser.nextToken() != JsonToken.END_ARRAY) {
                JsonNode node = parser.readValueAsTree();
                // before adding the document, it needs to be preprocessed
                loader.preprocessNode(node);
                docs.add(Document.parse(node.toString()));
            }
        }
    }
}
```

Figure 7.5: The method for parsing the JSON file using Jackson Streaming API

7.5 Model package

The model contains the domain classes and classes providing access to an SQL database. It also encapsulates all the domain logic in the application.

The model classes are called from the controllers which accept inputs

from the user interface. The model classes then do some operations, for example access the database, and either return results back to the original controller or set some of its properties which will be observed from the outside.

The Figure 7.6 shows the class diagram for the model package.

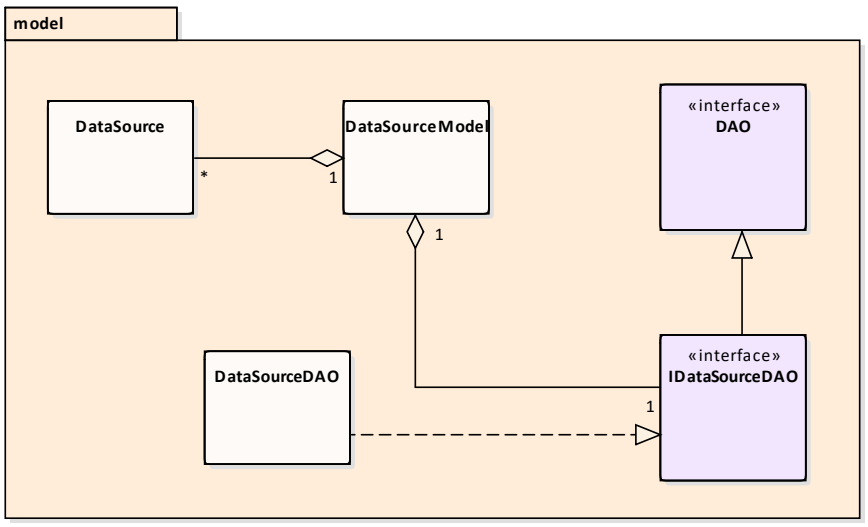


Figure 7.6: A class diagram of the model package

7.5.1 DataSource

`DataSource` is the domain object representing one source of data. For example PATSTAT database is one of the data sources. It contains various metadata about the data source.

The following properties are included in the `DataSource` object:

- id – Id of the data source. Corresponds to the id in the data sources database.
- name – Name of the data source.
- description – Description of the data source.
- url – URL path, where the data were acquired from.

- schemaPath – Path on the filesystem to the file of the schema of the data source. The schema contains all the fields that can appear in the particular data source.
- mappingPath – Path to the mapping file.
- licenceType – Type of license for that data source. It can either be free or it can be a subscription-based data source.
- licencePath – Path to the license file.
- categoryType – Category of data that the data source holds. The options are publications or patents.
- dataLastUpdated – Date, when the data source entry has been last updated.
- updateInterval – Interval in which the user should be notified to update the data source.

7.5.2 DAO

A simple interface which provides methods to setup a connection to the database, connect to it and close the connection.

7.5.3 IDataSourceDAO

Another interface which inherits the `DAO` interface and extends it by methods necessary for manipulating with data sources in the database. It contains basic CRUD operations with the data sources, like inserting, updating and deleting. It contains additional methods like fetching all the records or getting available data category types.

7.5.4 DataSourceDAO

Concrete implementation of the `IDataSourceDAO`. To access the database, the Apache Commons DbUtils library was used. It is a small set of classes designed to make working with JDBC easier [15]. It is able to use parametrized prepared statements necessary to ensure security and prevent SQL injections. They are also faster, because the preparation of the query is done only once.

7.5.5 DataSourceModel

Handles domain logic. Contains methods for loading data from the SQL database. The currently selected item in the ListView is set to `ObjectProperty<DataSource>` object. All the data sources are retrieved using the `loadData()` method. The returned list is then set to the `ObservableList<DataSource>` list. The controllers then add a listener to this observable list, so that every time the selection changes, the currently selected property is set to that item. This results in different data being rendered in the user interface.

7.6 DbAccess package

The `DbAccess` package contains a single class `DbManager` which serves a purpose of establishing a connection to the database. The connection properties are read from a configuration file `mydb.cfg` which must be present in the same directory, from where we run the application. The connection is closed using the `closeConnection()` method.

7.7 Logging

Various events are logged using a custom logger which is defined in the class `MyLogger`. It sets up the logger and configures it. The logging is done to both to console and to the file, so previous logs are stored on the file system.

7.8 Testing

The testing of the data administration application was performed with unit tests and automated tests of the GUI. Basic operations on the SQL database were tested as well.

All the tests are present in the `MainTest` class.

7.8.1 Unit Tests

Firstly, some unit tests were created for testing basic functions. For the database, it was also appropriate to test the correctness of operations on the database. Series of test cases were created to test the common operations:

- `testDatabaseInsertRecord()` – Adds a new data source to the database.

- `testDatabaseRemoveRecord()` – Removing a data source from the database.
- `testDatabaseUpdateRecord()` – Updating an existing data source in the database.

7.8.2 GUI Tests

For testing the user interface, I used a library called *TestFX* which is designed to test JavaFX applications. It is supported since Java version 1.8 and provides a clean and fluent API for manipulating the user interface. The tests can then be viewed in real time and see exactly what is happening.

Before the start of the tests, all the JavaFX controllers are instantiated and all FXML files are added to the FXML loader. The main layout is created as well. Finally the JavaFX scene is created.

The tests are created in the class `MainTest` and some helper methods are in the class `TestHelper`.

The following tables 7.2, 7.3 and 7.3 show some of the test cases for testing the graphical user interface.

Test case ID	TC_01	
Test name	testFieldsEditableAfterEditButtonClicked	
Description	Tests if the fields are editable after clicking the edit button. Also checks if the appropriate buttons are enabled and visible.	
Pre-conditions	Some item in the list view is selected. Nothing in the GUI has been done since the start.	
Step	**Action**	**Expected result**
1	Assert that delete button is enabled	
2	Click on edit button	Save button and discard button should become enabled and all the text fields editable.
3	Assert that all the fields are editable, delete button is enabled and visible, save button is visible and disabled, discard button is visible and disabled and edit button is disabled.	
4	Click on discard button	See post-condition.
Post-condition	The state should be the same as in the beginning.	

Table 7.2: Test case 1

Test case ID	TC_02	
Test name	testFormValidatesUpdateIntervalIsNotNumber	
Description	Tests if the validation of the update interval field fails if the value is not a number.	
Pre-conditions	Some item in the list view is selected. Nothing in the GUI has been done since the start.	
Step	**Action**	**Expected result**
1	Click on 'Add new button'	An empty form appears in the details panel
2	Fill all the fields, type text into update interval field.	All the fields are filled.
3	Click on 'Save' button	An error dialog appears.
4	Assert that the alert text contains: 'The update interval is not a number.'	See post-condition
Post-condition	Validation fails and nothing is saved to the database	

Table 7.3: Test case 2

Test case ID	TC_03	
Test name	testFormValidatesSourceNameTextFieldEmpty	
Description	Tests if the validation of the 'sourceNameTextField' field fails if the value is not a number.	
Pre-conditions	Some item in the list view is selected. Nothing in the GUI has been done since the start.	
Step	**Action**	**Expected result**
1	Click on 'Add new button'	An empty form appears in the details panel
2	Fill all the fields, leave empty field 'sourceNameTextField'.	All the fields are filled except from 'sourceNameTextField'.
3	Click on 'Save' button	An error alert appears.
4	Assert that the alert text contains: 'Source name cannot be empty.'	See post-condition
Post-condition	Validation fails and nothing is saved to the database	

Table 7.4: Test case 3

8 API Server Application

This application's purpose is to create an access point to the database by providing a simple and unified API. The API will be used by clients, who will be able to send various queries and receive results back. The communication between the application and the client should work using a REST architecture.

This chapter will first outline the architecture of the REST API server application and then describe each class in more detail. To demonstrate the functioning of the API, a separate application was created. It is a web application built with JavaEE and will provide some simple user interface to display the results, received from the database. It will also allow for simple visualization of these results. The description of the demonstration application will be given at the end of this chapter.

8.1 Architecture

The architecture follows practices for creating RESTful APIs. A RESTful API is a technology based on REST architectural style which is often used for communication over the internet. It uses HTTP requests GET, PUT, POST and DELETE. It is a more lightweight technology in comparison to SOAP for example.

The advantage of REST is that it is stateless, therefore it can be freely redeployed if something fails and can scale to accommodate load changes [8].

The application is deployed to Tomcat server and uses a RESTEasy framework. RESTEasy is a JBoss implementation of Java API for RESTful Web Services (JAX-RS).

List of all the libraries used in the application can be seen below on table 8.1.

Name	**Version**
RestEasy JAX-RS	3.6.3
Gson	2.8.5
Mongo Java Driver	3.9.1

Table 8.1: Used libraries in the application

The Figure 8.2 shows the class diagram of the RESTful API application.

It consists of two main packages, `rest` package which contains service class for creating the definitions of the API methods. These methods communicate with the `core` package which handles the database querying and other logic.

8.1.1 Client-Server communication

The scenario of the flow of events during the communication between the client and the server can be seen on Figure 8.1.

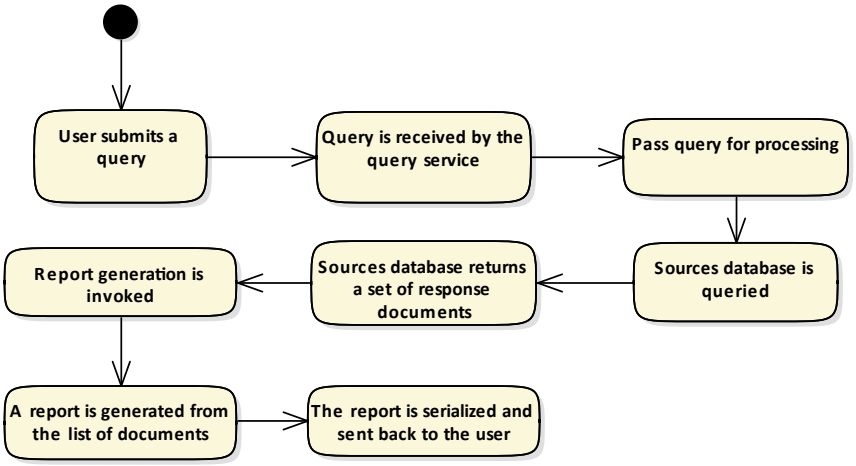


Figure 8.1: A flow of events for user when querying the database

The communication is initiated by the user, who submits a query and sends an HTTP request to the server. The query is deserialized on the server side in the `QueryRestService` class. The `Query` object is passed to the `DataRetrieval` class, where a MongoDB query is constructed and the sources database is queried. After the results are collected, `ReportCreator` class wraps the results set into a `Report` model class, where additional information is appended to the report. The report is then passed back to the `QueryRestService` class which serializes the results to JSON and sends them back to the client as a HTTP response body.

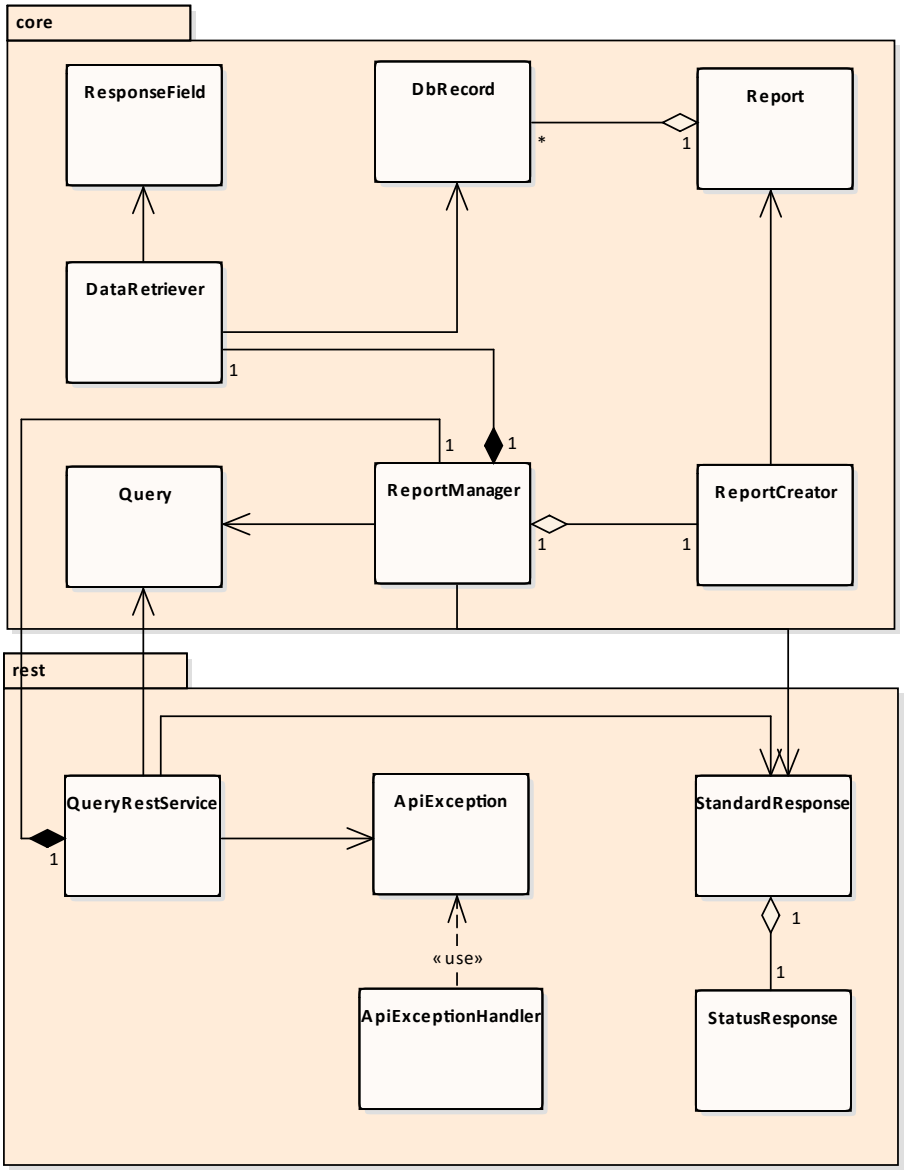


Figure 8.2: A class diagram of the REST Server application

8.1.2 Query

The query class represents a single query, received as body of a request. Each time a query is called on the API, new query object is constructed by deserializing the data content of the request from JSON format.

It contains the following fields:

- String dataSouce – Source of data we want to search in. Possibilities are *patents* and *publications*.

- String filter – Specifies fields, in which we want to search.
- String query – The actual text representation of the query.

8.1.3 QueryRestService

This is the main API service class. It contains definition of methods of the API. It accepts incoming requests and routes them forward to the classes in the `core` package. The concrete specifications of the methods will be described in the next section 8.2.

8.1.4 DataRetriever

The `DataTertiever` is responsible for communication with the sources database. It conststructs queries to MongoDB by utilizing the provided API of the MongoDB Java driver and returns a set of results.

Method `doSearch()` constructs a MongoDB query and runs it on a MongoDB collection. There are two types of searches: Full-text search which will return results from all the fields which are included in the text index. The second search option is a regular expression search which will search in specified fields, with the cost of slower performance. (see 5.6.2).

The query is constructed using a filter which determines if we are going to search textually or using a regular expression.

On line 5, the pagination is implemented using the `skip` method of MongoDB, where the range of results is calculated using the passed page parameter. We limit the returned results using the `limit` method. Next, we specify a projection which will ensure only the necessary fields will be present in the final result set. Finally the results are sorted according to their text score.

The code listing 8.3 below shows the construction of a query which will be passed to MongoDB collection.

```
1   private List<DbRecord> doSearch(Bson filter,
        MongoCollection<Document> collection, int limit, int
        page){
2       List<DbRecord> dbRecords = new ArrayList<>();
3       MongoCursor<Document> cursor;
4       cursor = collection.find(filter)
5               .skip(page > 0 ? ((page-1) * limit) : 0)
6               .limit(limit)
7               .projection(fields(Projections.metaTextScore(
                    "score"), include(
8                       ResponseField.TITLE.toString(),
9                       ResponseField.YEAR.toString(),
10                      ResponseField.ABSTRACT.toString(),
11                      ResponseField.AUTHORS.toString(),
12                      ResponseField.OWNERS.toString(),
13                      ResponseField.DOCUMENT_ID.toString(),
14                      ResponseField.DATA_SOURCE.toString())
                            ))
15              .sort(Sorts.metaTextScore("score"))
16              .iterator();
17  }
```

Figure 8.3: The method for construction of a MongoDB query

8.1.5 DbRecord

The results from the Mongo database are wrapped in a `DbRecord` class.

8.1.6 Report

Contains the returned data ready to be sent to the client. It provides a `getAsJson()` method which is used to serialize the `Report` object into JSON (see 8.2.3).

8.1.7 StandardResponse

This class holds all the data for a response. That includes the whole report plus some additional fields, like number of searched records or number of returned results.

8.2 REST API Specification

As was already described, the application will provide a REST API which will be used by the clients. The primary use of the REST API will be to consume queries submitted by the users on the client side and provide a response to that query back to the client.

8.2.1 Query Endpoint

The main method of the API is the query endpoint:

- **Description** Provides method to call a query and get a result set as a response.
- **URL**

  `/query/:page`

- **Method**

  `POST`

- **URL Params**

  Required:

  `page=[integer]`

- **Data Params**

```
{
    "sourceType": "patent",
    "filter": "",
    "query": "electric car"
}
```

- **Success Response** The success response can be seen in the Listing 8.4.
- **Error Response**
  - **Code:** 404 NOT FOUND

8.2.2 Request Format

The request format holds the serialized `Query` class in JSON format:

```
{
    "sourceType": "patent",
    "filter": "",
    "query": "electric car"
}
```

8.2.3 Response Format

The format of the response can be seen below.

The fields in the response have the following meaning:

- *msg* – The message associated with the response
- *status* – Status of the response. Can be either SUCCESS or ERROR. In case of an error status field, the msg field will contain information about the error.
- *searchedCount* – The number of searched documents.
- *returnedCount* – Number of returned documents.
- *reportJson* – Contains a list of documents relevant to the query (Report).

8.3 Database Testing

To demonstrate the created REST API and to test the database, a separate web application was created. It is built using Java servlets in Java EE 7 and JavaServer Pages (JSP) technology. The communication with the server is assured using a REST framework Jersey which simplifies the development of the RESTful clients in Java and provides some additional API functions on top of the JAX-RS framework. The demonstration application also allows for simple visualization of query results.

The web interface, seen on Figure 8.5, provides a simple layout with the ability to perform queries and display the results in a clear way. An HTML, CSS and JavaScript library Bootstrap was used to create the interface.

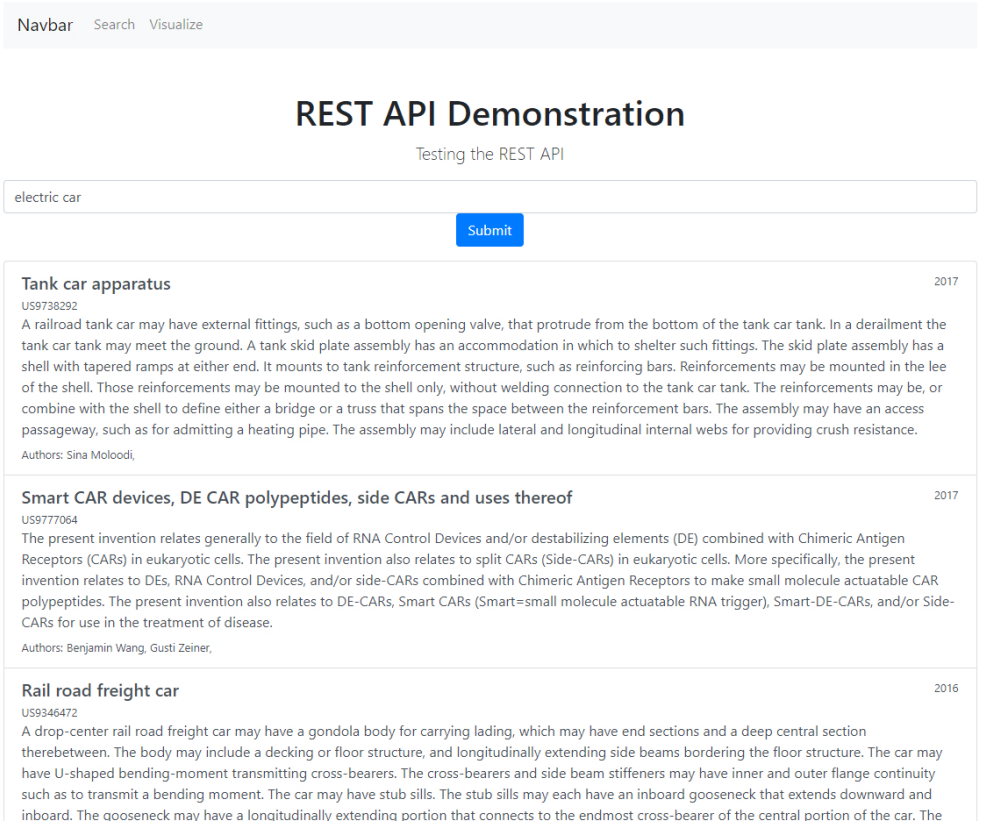


Figure 8.5: A screenshot of the REST API demonstration web application

List of all the libraries used in the application can be seen below on table 8.2.

Name	**Version**
Java servlet API	3.1.0
JSTL	1.2
Jersey Client	2.28
Tablesaw	0.32.6
JSoup	1.11.3

Table 8.2: Used libraries in the application

The class diagram on Figure 8.6 shows created classes and their relationships. There are two servlets, `QueryServlet` for handling requests from the `query.jsp` page and `VisualizeServlet` from the `visualize.jsp` page.

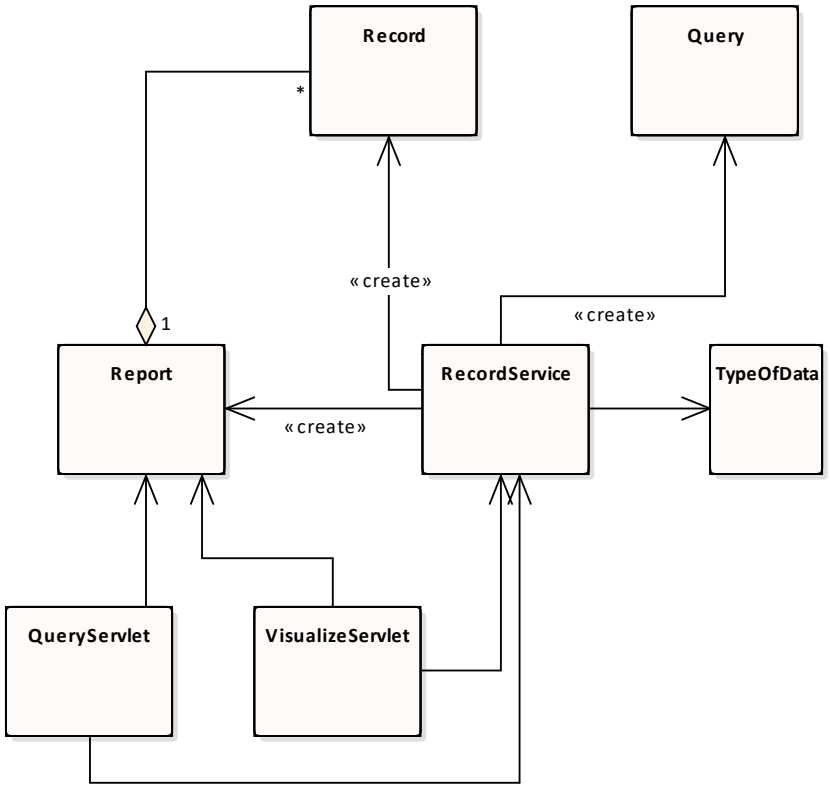


Figure 8.6: Class diagram of the REST demonstration application

8.3.1 RecordService class

The `RecordService` encapsulates the REST client. It contains a `fetchReports()` method which creates a REST request from the user query with the required format and sends it to the REST server. The report returned as a response from the server is manually deserialized into the `Report` class and it is returned back to the servlet to be visualized in the jsp page.

8.3.2 QueryServlet

The query servlet receives requests from the `query.jsp` web page's form and extracts the necessary information about the query. The `fetchReport()` method of the `RecordService` class is called to fetch the report.

8.3.3 VisualizeServlet

The servlet uses *tablesaw* Java visualization library which supports visualization of data by providing a wrapper to the *Plot.ly* javascript library.

It also uses the `fetchReport()` method of the `RecordService` class to fetch the report, but just for visualization purposes, another REST method was created which has almost the same signature as the `query` endpoint, but has one more parameter which is the number of documents we want to return. Therefore to visualize the report, we chose to return 1000 documents. For that we construct a graph showing how many documents were published each year. An example graph can be seen below on Figure 8.7.

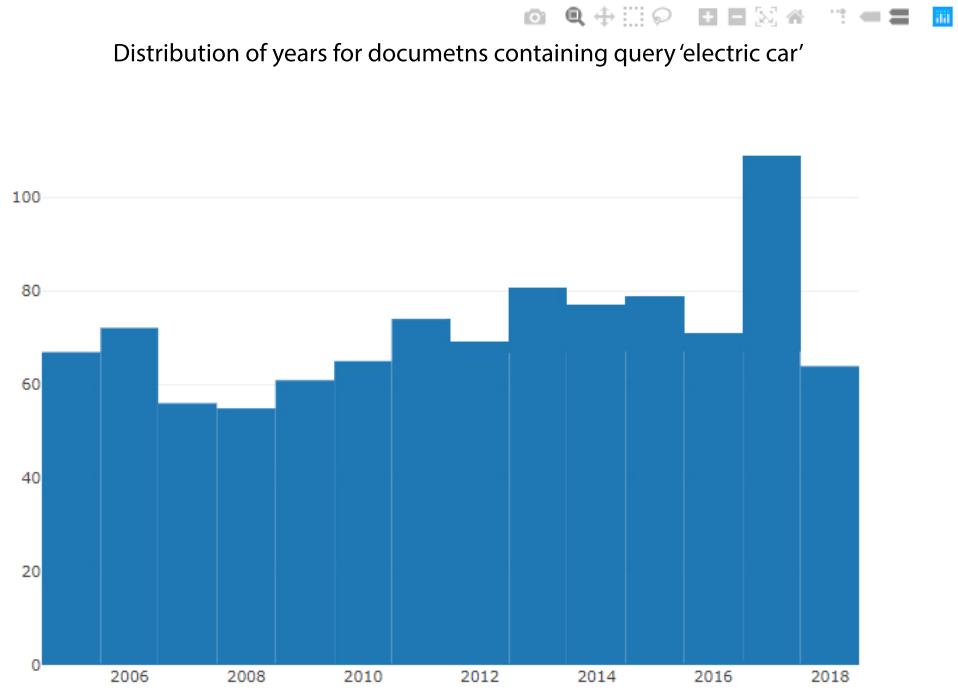


Figure 8.7: Visualization of years in which patents were published containing query 'electric car'

8.4 Future work

The development will grow beyond the scope of this thesis. The created solution is working, but can be much improved in the future.

Firstly, it will be necessary to identify and acquire additional data sources. Every data source will then have to be loaded to the sources database. Data

sources like Springer LOD or PATSTAT have been chosen to be the next sources of data and will be included in the main database as well. It is possible that some data from different sources can overlap, therefore it will be necessary to identify duplicates.

The deployment of the application will have to be considered in the future as well. The final solution should run on a production server, accessible over the internet.

So far, the REST API contains only a single method, query. In the near future, it will be necessary to expand it to provide additional functionalities, like returning a list of authors or owners which are present in the data.

```
{
   "msg":"Everything OK",
   "status":"SUCCESS",
   "searchedCount":3645421,
   "returnedCount":30,
   "reportJson":{
      "summary":"Test summary",
      "documents":[
         {
            "_id":{
               "$oid":"5c8e7d9cfa6e572f9cecea94"
            },
            "abstract":"A railroad tank car may have external
                fittings, such as a bottom opening valve,
               that protrude from the bottom of the tank car
               tank. In a derailment the tank car tank may
               meet the ground. A tank skid plate assembly
               has an ...",
            "number":"US9738292",
            "authors":[
               {
                  "name":"Sina Moloodi"
               }
            ],
            "owners":[
               {
                  "name":"NATIONAL STEEL CAR LIMITED"
               }
            ],
            "title":"Tank car apparatus",
            "year":"2017",
            "dataSource":"uspto"
         }
      ]
   }
}
```


Figure 8.4: The sample response to a query REST request

9 Measurements

An analysis of each data source was created, gathering various statistical data from the data sets. The approach for acquiring the statistical data was to create specific queries to the MongoDB database which return the relevant data in a suitable format for visualization.

This chapter will focus on the acquired data sets as well as their statistical data. It will also compare the performance of different queries to the MongoDB database.

9.1 Data

Table 9.1 shows the acquired data sources, their size and time range:

Source	Extracted size (GB)	Count	Time Range (years)
MAG	249	166m	1998 – 2017
USPTO	75.7	3.7m	2005 – 2018
DBLP	2.24	4.5m	1995 – 2018
PATSTAT	1	120	-
Total	327.94	174m	

Table 9.1: Acquired data

One of the biggest patent databases available today is the PATSTAT and therefore it would be beneficial to include it in the main database. However, to get data from PATSTAT database, it is necessary to pay a subscription fee. Unfortunately, at the time of writing this thesis, it was not yet purchased and only a sample files were available for download. However, the process of loading the data to the main database was prepared in advance, so in the future it will be faster and easier to load the PATSTAT data to the main database as well.

9.1.1 Query creation

Multiple MongoDB queries were created to extract relevant information about each data source. The queries were created using MongoDB's aggregation framework which were optimized for performance. The following code listing illustrates a sample MongoDB query:

The example query below searches for the most active authors of patents.

```
db.patent.aggregate([
  {$project: { _id: 0, "authors.name": 1 } },
  {$unwind: "$authors" },
  {$group: { _id: { $toLower: "$authors.name" }, count: {
     $sum: 1 } }},
  {$project: { _id: 0,"authors.name": "$_id", count: 1 } },
  {$sort: { count: -1 } }
], { allowDiskUse: true })
```

This MongoDB query uses its aggregation pipeline and comprises of the following stages:

- Project the fields we want
- Unwind the authors array so we now have a record for every array element in every document
- Group on the author's name from the expanded documents
- Project into a document format you can use as group messed around with _id
- Sort the results in reverse order to see the the most active authors first

9.1.2 USPTO Aanalysis

The United States Patent and Trademark Office gathers huge amount of data about patents issued in the United States.

The global statistics of the USPTO data set can be seen below on table 9.1.2.

Patents	3 727 260
Authors	771 764
Owners	990 969

Table 9.2: Global statistics of the USPTO data

It is interesting to see who are the authors holding the most patents. Each patent can specify multiple inventors. An inventor is the individual, who created the patent. Then there are owners/assignees of patents. These are typically inventors' employer company which hold the rights to the patent. There can be one or more owners of a patent.

The following table 9.1.2 shows the 20 most active patent authors in the USPTO data.

Position	**Author's name**	**Count**
1	Shunpei Yamazaki	2333
2	Kangguo Cheng	1373
3	Roderick A. Hyde	1311
4	Lowell L. Wood, Jr.	1281
5	Jonathan P. Ive	962
6	Clarence T. Tegreene	785
7	Bartley K. Andre	758
8	Jordin T. Kare	752
9	Duncan Robert Kerr	740
10	Christopher J. Stringer	738
11	Richard P. Howarth	728
12	Daniele De Iuliis	704
13	Daniel J. Coster	701
14	Eugene Antony Whang	694
15	Matthew Dean Rohrbach	694
16	Peter Russell-Clarke	679
17	Hanbyul Seo	675
18	Ali Khakifirooz	658
19	Jody Akana	658
20	Alexander Reznicek	657

Table 9.3: List of the most active patent authors

From the acquired USPTO data, we can see that the most productive author of patents is a Japanese inventor in computer science field Shunpei Yamazaki with 2 333 patents. He is followed by Chinese inventor from IBM Kangguo Cheng, who claimed 1 373 patents since 2005. The close third place is held by Roderick A. Hyde with 1311 patents.

Another interesting data to see is the distribution of institutions according to the number of patents ownership. The concrete numbers are seen on the table 9.4.

Position	Patents owner	Count
1	International Business Machines Corporation	29033
2	Samsung Electronics Co., Ltd.	21164
3	CANON KABUSHIKI KAISHA	10781
4	Apple Inc.	7781
5	QUALCOMM Incorporated	7771
6	Google Inc.	7447
7	LG ELECTRONICS INC.	7092
9	Intel Corporation	6389
11	Amazon Technologies, Inc.	5269
12	Ford Global Technologies, LLC	4885
13	Sony Corporation	4715
14	General Electric Company	4696
15	FUJITSU LIMITED	4378
16	Kia Silverbrook	4153
17	Huawei Technologies Co., Ltd.	3998
18	Samsung Display Co., Ltd.	3974
19	Taiwan Semiconductor Manufacturing C., Ltd.	3722
20	Hyundai Motor Company	3540

Table 9.4: List of the most active patent owners

The list of companies with the most patents is not surprising. The top institution which gained the most patents ownership, was IBM with 29 033 received patents. IBM holds the top position in the number of received patents for 25th consecutive time, when only in 2017 they received more than 9043 patents. Samsung Electronics Co. comes in second place with 21 164 received patents. In third place comes Canon which gained 10781 patents followed by Apple Inc. with 7781. The data contains patent information since from the year 2005 since the data were collected.

9.1.3 MAG Aanalysis

MAG is a graph comprising of more than 120 million publication entries. The data set contains fields like author names, institutions, fields of study etc. The global statistical information are seen on the table below.

Papers	123 000 000
Authors	25 828 122
Fields of study	47 989

Table 9.5: Global statistics of the MAG data source

The following table shows the most active authors of publications 9.6 in the MAG data:

Position	**Author**	**Count**
1	佐藤	49891
2	木	47973
3	田中	41891
4	高	38172
5	小林	33668
6	Helmut Herrmann	33231
7	Herbert Bucksch	33183
8	中村	33183
9	伊藤	32362
10	山本	30781
11	加藤	27854
12	渡	26314
13	吉田	26003
14	山田	25434
15	Wei Wang	24225
16	Jornal da Manhã	24073
17	Richard J. Lewis	22917
18	井上	22102
19	Cristiano da Silva Teixeira	20415
20	Wei Zhang	19972

Table 9.6: List of the most active publication authors

The publication's fields of study are an important information. MAG provides a set of study fields as a part of every publication. The Figure 9.1 illustrates the distribution of papers according to the field(s) of study they belong to.

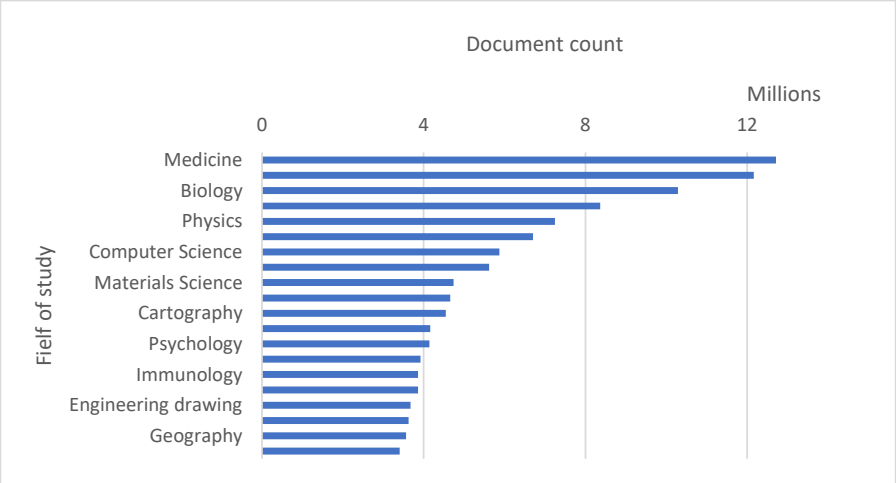


Figure 9.1: Document counts according to their field of study

We can see that the most prominent fields of study in the acquired data set from MAG are medicine, biology and physics, followed by computer science.

Then there is the year of a publication, another very important field. Histogram of years of publication provided in the MAG can be seen below on 9.2:

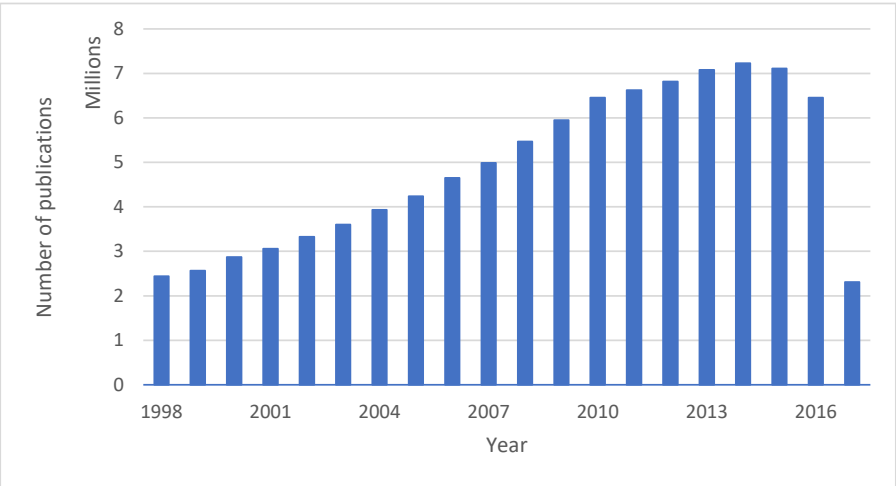


Figure 9.2: Histogram of years of publications in the MAG

9.2 Performance Measuring

9.2.1 Execution Environment

All the described applications and databases were created and run on the computer with the following hardware and software specifications (9.7).

Processor	Intel(R) Core(TM) i7-7700K CPU @ 4.20GHz
Video Card	Intel(R) HD Graphics 630
Video Card #2	NVIDIA GeForce GTX 1050 Ti
RAM	64 GB
Operating System	Windows 10
MongoDB db version	v4.0.5
Java	1.8

Table 9.7: HW and SW Specification of the computer

9.2.2 Experiments

One of the most important features of a database is the speed with which it can search for data. To get the best performance in the MongoDB database, it was necessary to create appropriate indexes on the fields on which the searches will be performed. As described in 3.1, Mongo provides two types of indexes: standard indexes and text indexes. The text indexes are specifically used for full-text searching.

A series of test queries were performed to compare the performance of full-text searching in MongoDB. An example query searches for the phrase "electric car". MongoDB looks for all the documents which contain the term "electric" *OR* "car" in the fields contained in the text index. The first page of 10 results is returned after 4.5 seconds. After the second try when the results are loaded in the RAM, the time was 0.11 seconds. However, when performing a count query on the same phrase, MongoDB took considerably longer, 1438 seconds, because the entire collection has to be scanned to fetch all the satisfying results.

It is evident that the creation of appropriate indexes was essential for getting a reasonable performance on the MongoDB database.

Another important aspect of the database is the performance of inserting new data to the database. In case of USPTO, the average size of a file is 148 MB. There are 5119 records on average in each file. That means that an average space occupied by a document approximately 28.9 kB.

Because each document is being parsed from the file system to the memory and then preprocessed. The average time to parse the JSON file from the file system was 2.45 seconds, and the preprocessing took 0.12 seconds on average. MongoDB was able to insert 3230 documents to the collection per second. This means that an average speed of insertion was 94 MB per second.

MongoDB supports aggregation pipeline, the example query below counts all the authors in a collection. The test was run on two collections with dramatically different sizes. The results of the tests are visible on table 9.2.2:

```
db.patent.aggregate([
  {$project: { _id: 0, "authors.name": 1 } },
  {$unwind: "$authors" },
  {$group: { _id: "$authors.name", count: { $sum: 1 } }},
  {$match: {count: { $gt: 1 } } },
  {$count: "count"}
], { allowDiskUse: true })
```

Documents count	Time(seconds)
3.7 m	57.911
107 m	2586.587

Table 9.8:

To run this query on the first collection, MongoDB took 57.911 seconds. After that, the query was run once again, when the previous results were saved to RAM. The second time it took 48 seconds. The second, larger collection took considerably longer to execute. In the first try, it took 2586.587 seconds.

10 Conclusion

The NoSQL approach was chosen for storage of tens of millions of documents. MongoDB database was selected as a primary database for storing the acquired data.

To summarize the work done in this thesis, the first part of this paper outlined main differences between relational and non-relational databases, several NoSQL databases were described. The second part was concerned with the actual implementation. Firstly the process of gathering and acquiring data from various data sources was described. Then the steps for preprocessing the data were presented. These steps included conversion of data to JSON format, a necessary step because MongoDB stores data in JSON(internally BSON). Another key part of the data preprocessing was the creation of a method to unify the structure of various data sources by designing and creating mapping files. This enabled altering the structure of the source JSON documents. Finally the process of loading the data to the primary database was presented. The database was then tested using a demonstration web application. It later showed that there were multiple inconveniences while working with MongoDB. These included especially lack of better full-text searching capabilities, like support for conditional queries and the ability to perform searches only in specified fields. In the last part of the thesis, a simple analysis of some data sources was done to give more insight about the type of data being stored.

Over 200 millions of records were acquired from multiple publicly available data sources, with Microsoft Academic Graph and United States Patent and Trademark Office being the most dominant. The created software solution consists of several components: The NoSQL MongoDB database, the SQL database for storing meta-data about data sources, data administration application, REST API server application and web client application. The whole system enables storing large amounts of data from various sources, administering the data sources and testing the created REST API.

Even though the main focus of this work was mainly data gathering, it later turned out that the spectrum of things needed to do was much more diverse than initially thought. That also resulted in the final implementation being more broad rather than in depth and some of the planned features were not implemented in the final solution. However, the development of this project will continue further in the future beyond the scope of this thesis with more features planned to be implemented.

Appendixes

Sample MongoDB Queries

The following are some of the queries used to create the statistics of data sets.

Searches for the most active authors of patents.

```
db.patent.aggregate([
  {$project: { _id: 0, "authors.name": 1 } },
  {$unwind: "$authors" },
  {$group: { _id: { $toLower: "$authors.name" }, count: {
      $sum: 1 } }},
  {$project: { _id: 0,"authors.name": "$_id", count: 1 } },
  {$sort: { count: -1 } }
], { allowDiskUse: true })
```

Count of authors in the collection:

```
db.patent.aggregate([
  {$project: { _id: 0, "authors.name": 1 } },
  {$unwind: "$authors" },
  {$group: { _id: "$authors.name", count: { $sum: 1 } }},
  {$match: {count: { $gt: 1 } } },
  {$count: "count"}
], { allowDiskUse: true })
```

Data Download URLs

The following list of URLs, from where the data sources were acquired:

- PATSTAT – https://www.epo.org/searching-for-patents/business/patstat.html#tab-1
- DBLP – https://dblp.uni-trier.de/faq/How+can+I+download+the+whole+dblp+dataset
- USPTO – https://bulkdata.uspto.gov/
- MAG – https://aminer.org/open-academic-graph

USPTO Sample Data

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE us-patent-grant SYSTEM "us-patent-grant-v42-2006-08-23.dtd" [ ]>
<us-patent-grant lang="EN" dtd-version="v4.2 2006-08-23" file="USD0673346-20130101.XML" status="PRODUCTION" id="us-patent-grant" country="US" date-produced="20121217" date-publ="20130101">
 <us-bibliographic-data-grant>
     <publication-reference>
         <document-id>
             <country>US</country>
             <doc-number>D0673346</doc-number>
             <kind>S1</kind>
             <date>20130101</date>
         </document-id>
     </publication-reference>
     <application-reference appl-type="design">
         <document-id>
             <country>US</country>
             <doc-number>29408358</doc-number>
             <date>20111212</date>
         </document-id>
     </application-reference>
     <us-application-series-code>29</us-application-series-code>
     <us-term-of-grant>
         <length-of-grant>14</length-of-grant>
     </us-term-of-grant>
     <classification-locarno>
         <edition>9</edition>
         <main-classification>0207</main-classification>
     </classification-locarno>
     <classification-national>
         <country>US</country>
         <main-classification>D 2639</main-classification>
     </classification-national>
     <invention-title id="d2e53">Belt end strap</invention-title>
     <references-cited>
         <citation>
             <patcit num="00001">
                 <document-id>
                     <country>US</country>
                     <doc-number>594201</doc-number>
                     <kind>A</kind>
                     <name>Prothingham</name>
```

```
                        <date>18971100</date>
                    </document-id>
                </patcit>
                <category>cited by examiner</category>
                <classification-national>
                    <country>US</country>
                    <main-classification>  2322</main-
                        classification>
                </classification-national>
            </citation>
        </references-cited>
        <number-of-claims>1</number-of-claims>
        <us-exemplary-claim>1</us-exemplary-claim>
        <us-field-of-classification-search>
            <classification-national>
                <country>US</country>
                <main-classification>D 2624-640</main-
                    classification>
                <additional-info>unstructured</additional-
                    info>
            </classification-national>
        </us-field-of-classification-search>
        <figures>
            <number-of-drawing-sheets>3</number-of-drawing-
                sheets>
            <number-of-figures>9</number-of-figures>
        </figures>
        <parties>
            <applicants>
                <applicant sequence="001" app-type="applicant
                    -inventor" designation="us-only">
                    <addressbook>
                        <last-name>Kosh</last-name>
                        <first-name>Matthew</first-name>
                        <address>
                            <city>Seattle</city>
                            <state>WA</state>
                            <country>US</country>
                        </address>
                    </addressbook>
                    <nationality>
                        <country>omitted</country>
                    </nationality>
                    <residence>
                        <country>US</country>
                    </residence>
                </applicant>
            </applicants>
```

```
            <agents>
                <agent sequence="01" rep-type="attorney">
                    <addressbook>
                        <orgname>Kilpatrick Townsend &#x26;
                           Stockton LLP</orgname>
                        <address>
                            <country>unknown</country>
                        </address>
                    </addressbook>
                </agent>
            </agents>
        </parties>
        <assignees>
            <assignee>
                <addressbook>
                    <orgname>Bodypoint, Inc.</orgname>
                    <role>02</role>
                    <address>
                        <city>Seattle</city>
                        <state>WA</state>
                        <country>US</country>
                    </address>
                </addressbook>
            </assignee>
        </assignees>
        <examiners>
            <primary-examiner>
                <last-name>Nelson</last-name>
                <first-name>T. Chase</first-name>
                <department>2914</department>
            </primary-examiner>
            <assistant-examiner>
                <last-name>Sims</last-name>
                <first-name>Kathleen M</first-name>
            </assistant-examiner>
        </examiners>
    </us-bibliographic-data-grant>
</us-patent-grant>
```

Microsoft Academic Graph Sample Data

```
{
  "id": "53e9ab9eb7602d970354a97e",
  "title": "Data mining: concepts and techniques",
  "authors": [
    {
```

```
      "name": "jiawei han",
      "org": "department of computer science university of illinois at urbana champaign"
    },
    {
      "name": "micheline kamber",
      "org": "department of computer science university of illinois at urbana champaign"
    },
    {
      "name": "jian pei",
      "org": "department of computer science university of illinois at urbana champaign"
    }
  ],
  "year": 2000,
  "keywords": [
    "data mining",
    "structured data",
    "world wide web",
    "social network",
    "relational data"
  ],
  "fos": [
    "relational database",
    "data model",
    "social network"
  ],
  "n_citation": 29790,
  "references": [
    "53e99ef4b7602d97027c2346",
    "53e9aa23b7602d970338fb5e",
    "53e99cf5b7602d97025aac75"
  ],
  "doc_type": "book",
  "lang": "en",
  "publisher": "Elsevier",
  "isbn": "1-55860-489-8",
  "doi": "10.4114/ia.v10i29.873",
  "pdf": "//static.aminer.org/upload/pdf/1254/370/239/53e9ab9eb7602d970354a97e.pdf",
  "url": [
    "http://dx.doi.org/10.4114/ia.v10i29.873",
    "http://polar.lsi.uned.es/revista/index.php/ia/article/view/479"
  ],
  "abstract": "Our ability to generate and collect data has been increasing rapidly. Not only are all of our
```

```
    business, scientific, and government transactions now
    computerized, but the widespread use of digital cameras,
     publication tools, and bar codes also generate data. On
     the collection side, scanned text and image platforms,
    satellite remote sensing systems, and the World Wide Web
     have flooded us with a tremendous amount of data. This
    explosive growth has generated an even more urgent need
    for new techniques and automated tools that can help us
    transform this data into useful information and
    knowledge. Like the first edition, voted the most
    popular data mining book by KD Nuggets readers, this
    book explores concepts and techniques for the discovery
    of patterns hidden in large data sets, focusing on
    issues relating to their feasibility, usefulness,
    effectiveness, and scalability. However, since the
    publication of the first edition, great progress has
    been made in the development of new data mining methods,
     systems, and applications. This new edition
    substantially enhances the first edition, and new
    chapters have been added to address recent developments
    on mining complex types of data? including stream data,
    sequence data, graph structured data, social network
    data, and multi-relational data."
}
```

PATSTAT Sample Data

```
"application": {
        "application_id": 100008,
        "appln_id": 15706408,
        "appln_auth": "EP",
        "appln_nr": "00100008",
        "appln_filing_date": "2000-01-03",
        "filing_lg": "en",
        "status": 9,
        "internat_appln_id": 0,
        "internat_appln_nr": "",
        "status_text": "The application has been withdrawn"
      },
      "authors": [
        {
          "party": {
            "name": "Hitachi, Ltd.",
            "set_seq_nr": 1,
            "is_latest": "N",
            "change_date": "2000-06-09",
```

```
            "bulletin_year": 2000,
            "bulletin_nr": 30,
            "type": "A",
            "wishes_to_be_published": " ",
            "seq_nr": 1,
            "designation": "all",
            "customer_id": "0100140063",
            "address_1": "",
            "address_2": "",
            "address_3": "",
            "address_4": "",
            "address_5": "",
            "country": "JP"
          }
        },
        {
          "party": {
            "name": "Hitachi Engineering & Services Co., Ltd
                .",
            "set_seq_nr": 1,
            "is_latest": "N",
            "change_date": "2000-06-09",
            "bulletin_year": 2000,
            "bulletin_nr": 30,
            "type": "A",
            "wishes_to_be_published": " ",
            "seq_nr": 2,
            "designation": "all",
            "customer_id": "0100139895",
            "address_1": "",
            "address_2": "",
            "address_3": "",
            "address_4": "",
            "address_5": "",
            "country": "JP"
          }
        }
      ],
      "title": [
        {
          "title": {
            "title": "Electric power variation compensating
                device",
            "change_date": "2000-06-09",
            "bulletin_year": 2000,
            "bulletin_nr": 30,
            "title_lg": "en"
          }
```

```
        }
      ],
      "abstract": [
        {
          "abstract": {
            "bulletin_year": 2000,
            "bulletin_nr": 30,
            "publn_auth": "EP",
            "publn_nr": "1022838",
            "publn_kind": "A2",
            "publn_date": "2000-07-26",
            "publn_lg": "en"
          }
        }
      ],
      "ipc": [
        {
          "ipc": {
            "ipc_text": "H02J3/38",
            "change_date": "2000-06-09",
            "bulletin_year": 2000,
            "bulletin_nr": 30
          }
        }
      ],
      "prior": [
        {
          "prior": {
            "change_date": "2000-06-09",
            "bulletin_year": 2000,
            "bulletin_nr": 30,
            "prior_seq_nr": 1,
            "prior_kind": "al",
            "prior_auth": "JP",
            "prior_nr": "19990014268",
            "prior_date": "1999-01-22"
          }
        }
      ],
      "designated_states": [
        {
          "designated_states": {
            "state_type": "EXT",
            "change_date": "9999-12-31",
            "bulletin_year": 0,
            "bulletin_nr": 0,
            "designated_states": "AL,LT,LV,MK,RO,SI"
          }
```

```
        },
        {
          "designated_states": {
            "state_type": "MEM",
            "change_date": "2000-06-09",
            "bulletin_year": 2000,
            "bulletin_nr": 30,
            "designated_states": "AT,BE,CH,CY,DE,DK,ES,FI,FR,
               GB,GR,IE,IT,LI,LU,MC,NL,PT,SE"
          }
        }
      ],
      "dates": [
        {
          "dates": {
            "bulletin_year": 2004,
            "bulletin_nr": 36,
            "date_type": "WDRWNA",
            "event_date": "2004-04-28",
            "cause_interruption": "NA",
            "converted_to_country": "  "
          }
        }
      ]
```

DBLP Sample Data

A sample document in DBLP looks like this:

```

<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE dblp SYSTEM "dblp.dtd">
<dblp>

[...]

<article key="journals/cacm/Gentry10" mdate="2010-04-26">
<author>Craig Gentry</author>
<title>Computing arbitrary functions of encrypted data.</
   title>
<pages>97-105</pages>
<year>2010</year>
<volume>53</volume>
<journal>Commun. ACM</journal>
<number>3</number>
```

```
<ee>http://doi.acm.org/10.1145/1666420.1666444</ee>
<url>db/journals/cacm/cacm53.html#Gentry10</url>
</article>

[...]

<inproceedings key="conf/focs/Yao82a" mdate="2011-10-19">
<title>Theory and Applications of Trapdoor Functions (
   Extended Abstract)</title>
<author>Andrew Chi-Chih Yao</author>
<pages>80-91</pages>
<crossref>conf/focs/FOCS23</crossref>
<year>1982</year>
<booktitle>FOCS</booktitle>
<url>db/conf/focs/focs82.html#Yao82a</url>
<ee>http://doi.ieeecomputersociety.org/10.1109/SFCS.1982.45</
   ee>
</inproceedings>

[...]

<www mdate="2004-03-23" key="homepages/g/OdedGoldreich">
<author>Oded Goldreich</author>
<title>Home Page</title>
<url>http://www.wisdom.weizmann.ac.il/~oded/</url>
</www>

[...]
</dblp>
```

User Manual

Data Administration Application

The application for administering data sources and loading new data sources into the main sources database. The image 10.1 bellow shows the GUI of the application.

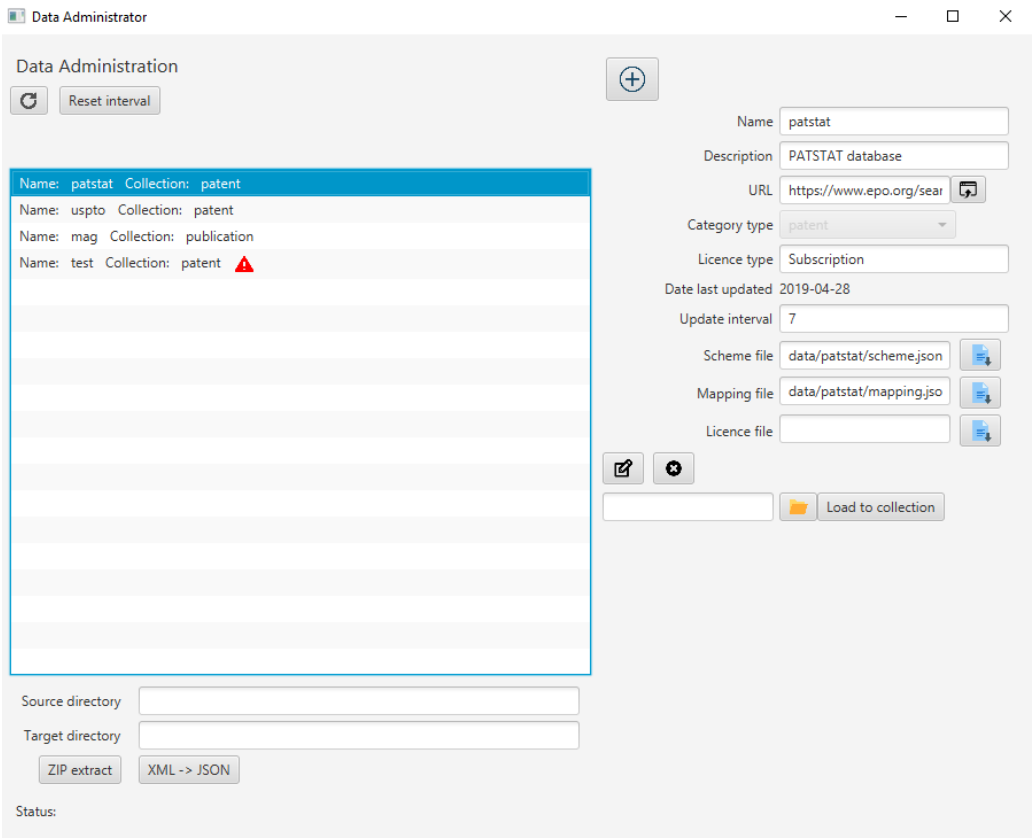


Figure 10.1: The GUI of the data administration application

The interface consists of a list view, containing a list of data sources metadata present in the Data Sources DB. On top of the list view, there is a *refresh* button which refreshes the list of data sources from the database. On the right side, there is the detail view for the selected item in the list view. It contains all the metadata stored for each data source. The details information about a data source can be edited using the *Edit* button and deleted using the *Delete* button. If the selected data source contains the implementation of loading it to the database, the *Load to collection* button attempts to load files from the file system path and insert it to the collection,

specified in the category type field. Each data field also contains information about its validity, specified by the update interval field. The update date can be updated using the *reset interval* button.

The bottom of the page contains utility tools for bulk zip extraction of files from a directory and bulk conversion of files from a source directory to the target directory.

At the very bottom, there is a label indicating the status of current operation.

Loading to the Database

For example, if we want to load data from the USPTO data source. We select the uspto list item from the list view on the left. We fill the path to the uspto JSON files to the text field next to the *Load to collection* button and finally press that button. The directory should be scanned and every JSON document should be loaded to the database.

Compilation

The compilation of the application is done using Maven's `assembly:single` goal[1]. To compile the project, navigate to the root directory of the project (`data-admin` folder) and double click on the build.xmland run the following command:

```
mvn clean compile assembly:single
```

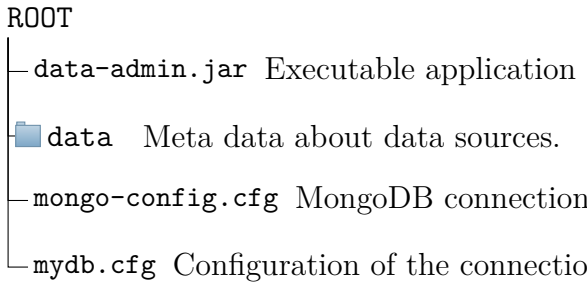
This command cleans all the files from previous compilations, starts a new compilation and then assembles the packaged executable. A `target` directory is created with the jar `data-admin.jar` inside.

For ease of use, a build script `build.bat` is present in the directory as well. Running it as an administrator will run the compilation process.

Execution Prerequisites

In order for the application to run, the following files need to be present in the directory with the JAR file or the application won't execute:

[1] In order for the compilation to run, Maven has to be present on the system and added to the system Path



The `data` folder contains mapping files necessary for loading the data sources to the sources database. `mongo-config.cfg` file specifies the MongoDB database, to which we want to connect. `mydb.cfg` contains configuration of the connection to the SQL database. The SQL database has to be created on the system and the database service needs to be running [2].

Execution

If all the prerequisites are met, the application can be started by double clicking the created JAR file or by running the following command from the command line in the directory with the JAR file:

```
java -jar data-admin.jar
```

[2] Script for the creation of the database is present on the attached CD.

CD Contents

The attached CD disk contains all the applications created along with an SQL database script and a sample dump of the Sources database. It also contains the text of the thesis in a pdf format and created poster. The image below shows the directory structure of the attached CD along with comments for each folder/file.

- `ROOT`
  - `sample-data` Sample data and metadata for several data sources.
  - `dump` MongoDB sample database dump.
  - `data-admin` Data Administration application
    - `data` Meta data about data sources
    - `doc` JavaDoc
    - `exe` Executable
    - `src` Source files
    - `build.bat` Build script
  - `db` SQL script for creation of MariaDB data sources database
  - `rest-client` REST client application
    - `doc` JavaDoc
    - `exe` Deployable war archive
    - `project` IntelliJ IDEA project
    - `src` Source files
    - `build.bat` Build script
    - `readme.txt`
  - `rest-server`
    - `...` Same structure as rest-client.
  - `text`
    - `thesis.pdf` Text of master thesis
  - `poster` Posters in Czech and English
  - `build-all.bat` Build script for building the all the applications

The `dump` folder contains dumped MongoDB sample database, containing

7000 patent documents and 150 000 publication documents [3]. To restore the database, use the following command:

```
mongorestore --db sources dump/
```

REST Client Deployment

The REST client application can be deployed to Tomcat server using the following steps:

- Compile and package the REST client using the provided build script in the `rest-client` folder[4] or use the already packaged file in the `rest-client/exe` folder.
- Move the created packaged war file into the `webapps` directory of the Tomcat installation.
- Run the `startup.bat` script file from the directory `"Tomcat installation directory"/bin`.
- In the browser, navigate to address `localhost:port/restTest`, where *restTest* is the name of the deployed war file.[5]

[3]The attached CD contains only the sample database, because of the CD's size limitation. The full database is present on a local machine on the Department of Computer Science and Engineering in University of West Bohemia.

[4]In order for the build scripts to work, maven has to be installed on the system and must be set in the system path.

[5]The REST client application expects that the REST server runs on address: `localhost:8080/`.

Glossary

BSON

A binary-encoded serialization of JSON-like documents. 6

FXML

A human-readable data serialization language, usually used for configuration files. 49

Intellectual property right

A right that is had by a person or by a company to have exclusive rights to use its own plans, ideas, or other intangible assets without the worry of competition, at least for a specific period of time. 1

NoSQL database

NoSQL is a non-relational database that stores and accesses data using key-values. Instead of storing data in rows and columns like a traditional database. 44

Relational database

A collection of data items with pre-defined relationships between them. These items are organized as a set of tables with columns and rows. 34

REST

An architectural style for developing web services. 43

RESTful API

Web service APIs that adhere to the REST architectural constraints. 13

XML

A markup language much like HTML. XML was designed to store and transport data and to be self-descriptive and readable by machine and human. 28

YAML

A human-readable data serialization language, usually used for configuration files. 6

Acronyms

ACID	Atomicity, consistency, isolation, durability 3
API	Application programming interface 61
CQL	Cassandra Query Language 12
CSV	Comma-separated values 28
DAO	Data access object vii, 57
GUI	Graphical user interface ix, 49
HTTP	Hypertext Transfer Protocol 61
JAX-RS	Java API for RESTful Web Services 61, 67
JDBC	Java Database Connectivity 57
JSON	Javascript object notation 6
JSP	JavaServer Pages 67
MVC	Model-view-controller 48
NoSQL	Not only SQL 3
OLTP	Online transaction processing 15
RDBMS	Relational database management system 2
REST	Representational state transfer 61
SOAP	Simple Object Access Protocol 61
SQL	Standard query language 3
USPTO	United States Patent and Trademark Office 24, 34
XML	eXtensible Markup Language 6

Bibliography

[1] Kavita Bhamra. *A Comparative Analysis of MongoDB and Cassandra*. PhD thesis, Department of Informatics University of Bergen, 2017.

[2] Scott Carlson. Challenging google, microsoft unveils a search tool for scholarly articles. *The Chronicle of higher education*, 52(33), 2006.

[3] Ramez Elmasri and Sham Navathe. *Fundamentals of database systems*. Pearson, 2014.

[4] Clinton Gormley and Zachary Tong. *Elasticsearch: The Definitive Guide*. 2015.

[5] Michael Gusenbauer. Google scholar to overshadow them all? comparing the sizes of 12 academic search engines and bibliographic databases. *Scientometrics*, 118(1):177–214, Jan 2019. ISSN 1588-2861. doi: 10.1007/s11192-018-2958-5. URL https://doi.org/10.1007/s11192-018-2958-5.

[6] Guy Harrison. *Fundamentals of database systems*. Apress, 2015. ISBN 9781484213292 1484213297. URL https://www.amazon.com/Next-Generation-Databases-NoSQLand-Data/dp/1484213300.

[7] Hugh MacMullan. Worldwide patents database: Patstat. https://research-it.wharton.upenn.edu/news/worldwide-patents-database-patstat/, 2016. Accessed 26.03.2019.

[8] Ed Hannan Margaret Rouse and Sarah Wilson. Restful api. https://searchmicroservices.techtarget.com/definition/RESTful-API, 2014. Accessed 04.04.2019.

[9] Nishant Neeraj. *Mastering Apache Cassandra*. Packt Publishing, 2nd edition, 2015. ISBN 1784392618, 9781784392611.

[10] NoSQL. Types of noSQL databases, 2017. URL https://www.mongodb.com/scale/types-of-nosql-databases.

[11] José Luis Ortega. *Academic search engines: A quantitative outlook*. Elsevier, 1 edition, 2014. ISBN 9781843347910.

[12] A. Poulter and M. Drake. Encyclopedia of library and information science. *Marcel Dekker*, (1):389–396, 2003.

[13] David Pressman and Thomas Tuytschaevers. *Patent it yourself: your step-by-step guide to filing at the US Patent Office*. Nolo, 14 edition, 2016. ISBN 978-1413310580.

[14] Mitko Radoev. A comparison between characteristics of nosql databases and traditional databases. Technical report, Department of Information Technologies and Communications, Faculty of Applied Informatics and Statistics, University of National and World Economy, Sofia, Bulgaria, 2017.

[15] Matan Sarig. Commons dbutils: Jdbc utility component. https://commons.apache.org/proper/commons-dbutils/, 2017. Accessed 26.03.2019.

[16] Matan Sarig. Cassandra vs mongodb in 2018. https://blog.panoply.io/cassandra-vs-mongodb, 2018. Accessed 26.03.2019.

[17] Arnab Sinha, Zhihong Shen, Yang Song, Hao Ma, Darrin Eide, Bo-June Paul Hsu, and Kuansan Wang. An overview of microsoft academic service (mas) and applications. In *Proceedings of the 24th International Conference on World Wide Web*, pages 243–246, 2015. doi: 10.1145/2740908.2742839. URL http://dx.doi.org/10.1145/2740908.2742839.

[18] SQL. Relational data model in dbms: Concepts, constraints, example. URL https://www.guru99.com/relational-data-model-dbms.html.

[19] Dan Sullivan. *NoSQL for mere mortals*. Addison-Wesley, 2015.

[20] Jie Tang, Jing Zhang, Limin Yao, Juanzi Li, Li Zhang, and Zhong Su. Arnetminer: extraction and mining of academic social networks. In *Proceedings of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 990–998, 2008.

[21] Rik Van Bruggen. *Learning Neo4j*. Packt Publishing, Birmingham, 2014. ISBN 978-1-84951-716-4.

[22] Website without author. What is cassandra? http://cassandra.apache.org/, . Accessed 26.03.2019.

[23] Website without author. Inverted index. https://www.elastic.co/guide/en/elasticsearch/guide/current/inverted-index.html, . Accessed 11.04.2019.

[24] Website without author. Microsoft academic graph. https://www.microsoft.com/en-us/research/project/microsoft-academic-graph/, . Accessed 26.03.2019.

[25] Website without author. What is mongodb? https://www.mongodb.com/what-is-mongodb, . Accessed 26.03.2019.

[26] Website without author. The mongo shell. https://docs.mongodb.com/manual/mongo/, . Accessed 26.03.2019.

[27] Website without author. Mongodb system properties. https://db-engines.com/en/system/MongoDB, . Accessed 26.03.2019.

[28] Website without author. Text indexes. https://docs.mongodb.com/manual/core/index-text/, . Accessed 26.03.2019.

[29] Website without author. Espacenet patent search. https://worldwide.espacenet.com/, 2017. Accessed 26.03.2019.

[30] Website without author. January 2018 graph update. https://www.microsoft.com/en-us/research/project/academic/articles/january-2018-graph-update/, 2018. Accessed 26.03.2019.

[31] Website without author. Academic knowledge api. https://docs.microsoft.com/en-us/azure/cognitive-services/academic-knowledge/home, 2018. Accessed 26.03.2019.

[32] Website without author. Canadian intellectual property office. http://www.ic.gc.ca/eic/site/cipointernet-internetopic.nsf/eng/Home, 2019. Accessed 26.03.2019.

[33] Website without author. What do i find in dblp.xml? https://dblp.uni-trier.de/faq/16154937, 2019. Accessed 06.05.2019.

[34] Website without author. Statistics - records in dblp. https://dblp.org/statistics/recordsindblp, 2019. Accessed 26.03.2019.

[35] Website without author. About google patents. https://support.google.com/faqs/answer/7049585, 2019. Accessed 26.03.2019.

[36] Website without author. Neo4j system properties. https://db-engines.com/en/system/Neo4j, 2019. Accessed 26.03.2019.

[37] Website without author. Patstat. https://www.epo.org/searching-for-patents/business/patstat.html#tab-1, 2019. Accessed 26.03.2019.

[38] Website without author. Web of science platform: Introduction. http://clarivate.libguides.com/webofscienceplatform, 2019. Accessed 13.04.2019.