DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Enrichment of contact data using information from public APIs

Markus Ehringer

DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Enrichment of contact data using information from public APIs

Anreichern von Kontaktdaten durch Informationen von öffentlichen APIs

Author:	Markus Ehringer
Supervisor:	Prof. Dr. Florian Matthes
Advisor:	Patrick Holl
Submission Date:	16.04.2018

I confirm that this bachelor's thesis in informatics is my own work and I have documented all sources and material used.

Munich, 16.04.2018 Markus Ehringer

Acknowledgments

Abstract

Contact lists are rarely as complete and/or up-to-date as desired. However, many use cases do exactly require these two characteristics. Therefore, contacts need to be enriched with additional data. Social media/network websites are predestined data resources because of their huge number of users. This work conducts an analysis of possible sources and their APIs. Data from those websites is usually neither flawless nor completely up-to-date. Creating appropriate metrics and a system to merge multiple data sources into one contact is essential. A main part of this work is the implementation of a modular prototype which allows to easily add new data resources. The more sources the more data. Despite the '*garbage in, garbage out*'-principle, more data can be beneficial in determining its correctness. Additionally, an effective synchronization approach to keep the data up-to-date is introduced.

Contents

Part I.

Foundation

1. Introduction

1.1. Problem statement

Most people have a list of contacts stored in the cloud, on the smartphone or in some contact manager. Due to a lack of information and carefulness these lists are in general neither complete nor up-to-date, like in table 1.1:

First Name	Last Name	Organization	Age	E-Mail
Nick	Pisa	Google		nick.pisa@gmail.com
Robert	Tuner	BMW	45	
Michelle	Rosa	TUM	32	mr86@tum.de

Table 1.1.: Example: Incomplete contact data

Table 1.1 shows that the age of Nick and the e-mail address of Robert are missing. Very overgeneralized the idea is to fill these blanks and adjust incorrect information. Once enriched the table should look like this:

First Name	Last Name	Organization	Age	E-Mail
Nick	Pisa	Google	**23**	nick.pisa@gmail.com
Robert	Tuner	BMW	45	**rub.tuner@gmx.at**
Michelle	Rosa	**LMU**	32	mr86@tum.de

Table 1.2.: Example: Completed contact data

As we can see, Nick is 23 years old, the e-mail of Robert is rub.tuner@gmx.at and apparently Michelle is now working at the LMU.

1.2. Existing solutions

In fact for the business sector there are fine products which gather contact information and enrich the companies' contact data, but not for individuals. *Clearbit* [5] uses over

250 public and private sources to add information to one's contacts. *iSEEit* [26] is a commercial *costumer relationship management (CRM)* tool which has a built-in 'enrich contact'-function. It searches the web for useful links and lets the user decide which ones to add to a profile. *Freshsales* [14] is another CRM tool to enrich contacts. *Fullcontact* [15] comes closest to our idea. Nevertheless, the problem with *Fullcontact* as well as with *Clearbit* and *Freshsales* is that an e-mail is needed to enrich the contacts. However, we think given the name and the company it should be possible to find additional information for any contact. *pipl* [36] is a search engine that finds profiles of persons on multiple social networks. Instead of showing the gathered information in one view it only provides a list of links. *TSIMMIS* [45], *"The Stanford-IBM Manager of Multiple Information Sources"*, is a project of the Standford University in cooperation with IBM. It is a tool to facilitate the integration of heterogeneous information sources. The idea is quite interesting and similary to ours but far more abstract and generalized.[1]

1.3. Motivation

Knowing more about people does not only profit oneself but also the people you are connecting with.

The *SEBIS* chair of the *Technical University of Munich (TUM)* has a list of contacts (each contact has the attributes *first name*, *last name* and *organization*) more or less active in the blockchain ecosystem.[2] Using this list they want to find appropriate contact persons, invite people to events and improve personal connections on future events.

1.3.1. Assessment of appropriate contact persons

If, for instance, a technical expert for IOTA is needed, name and company might not be enough to decide if the person is suitable as a counterpart. More information is needed: a short description about that person or even better a list of keywords describing his key areas of expertise. This makes it possible to find appropriate contact persons for specific subjects. In addition, to get in contact with a specific company, finding the right person is crucial. Often we want to talk to a person with a certain role, e.g. a manager. Therefore, we need to know the role of a person in a company.

[1] Further information in the paper of Chawathe et al. [4].
[2] In the later chapters we will always refere back to this particular use case.

1.3.2. Invite people to events

In the case of the *SEBIS* chair, they are planing a lecture called "Blockchain-based Systems Engineering" in the future semesters. The goal is to find appropriate guest lecturers. Therefore, an e-mail, phone number or anything else to get in touch with the people is crucial.

1.3.3. Improve personal connection on future events

Creating long lasting connections with people works best in person. Therefore, knowing which conferences and events people attended and especially will attend in the future is necessary.

In order to fulfill the above requirements the contacts of the chair need to be enriched with additional data (in our case from online sources).

1.4. Research Approach

1.4.1. Research Questions

In this work we address the following research questions:

1. **Which online sources could provide public available information for automatic processing?**
   At the beginning we will evaluate which of the APIs out there would fit the needs of enriching contact data best. And not only 'Which websites are there?' but also 'What data can I get from them (in a legal way)'?

2. **How can the quality of a source's information/data be assessed?**
   Barely anyone is controlling information users enter online. As data may be out-of-date, incomplete and/or incorrect taking information of multiple sources we want to assess and improve data quality.

3. **How can the contact data, once enriched, be synchronized with the sources efficiently?**
   Due to the globalization, keeping correct information up-to-date is difficult but very important. Finding an efficient way to do so is therefore crucial to further secure data quality.

1.4.2. Approach

The first part of this work will be a profound analysis about general contact requirements, public available sources where to get contact data from and which information can actually be extracted automatically. This should answer the first research question. After the analyse we will extract data from appropriate APIs which will be used for the evaluation in chapter 7.
In the second part we will build a prototype called *gordias*, which takes the information of several APIs, transforms it and stores it in a *data warehouse* [24] - ready to be accessed via an API.

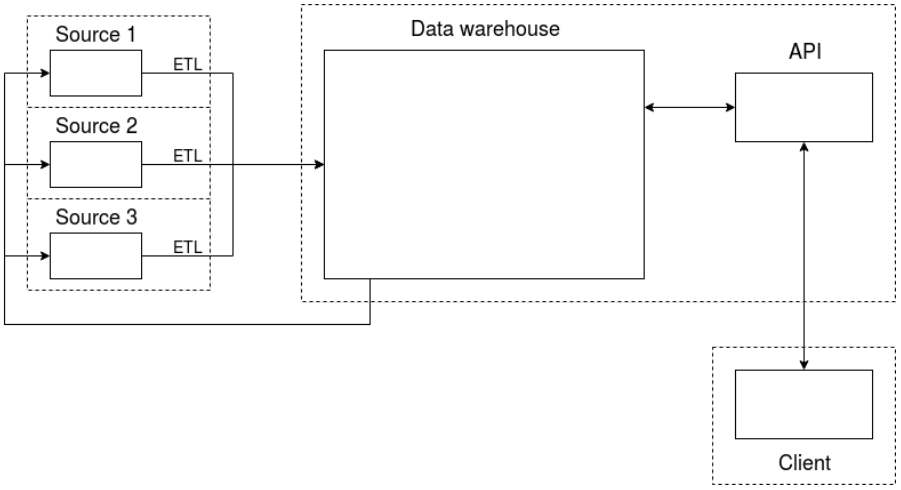

Figure 1.1.: General architecture

As we can see in figure 1.1 the client will send a request to the API. It does not matter if it is a single contact or a list of contacts. If contact information is available and sufficiently up-to-date (defineable by a request parameter - see section 4.2) it is fetched from the data warehouse, otherwise the data is requested from the source's APIs, stored in the database and then returned to the client. The prototype is organized into several modules. The main module consists of the internal API and the data warehouse, which itself consists of a database and logic for transforming data from the source's APIs and syncing data with the them. There will be *n* data-source modules which consist of the logic to fetch data from APIs and preprocess it for further usage in the data warehouse. Through an generic interface implementation new modules can be added at any time.

The figure 1.2 represents the concrete use case of the *SEBIS* chair. The 'client' consists of two parts: a tool called *midas* and *Socio Cortex (SC)*, which is the platfrom where the

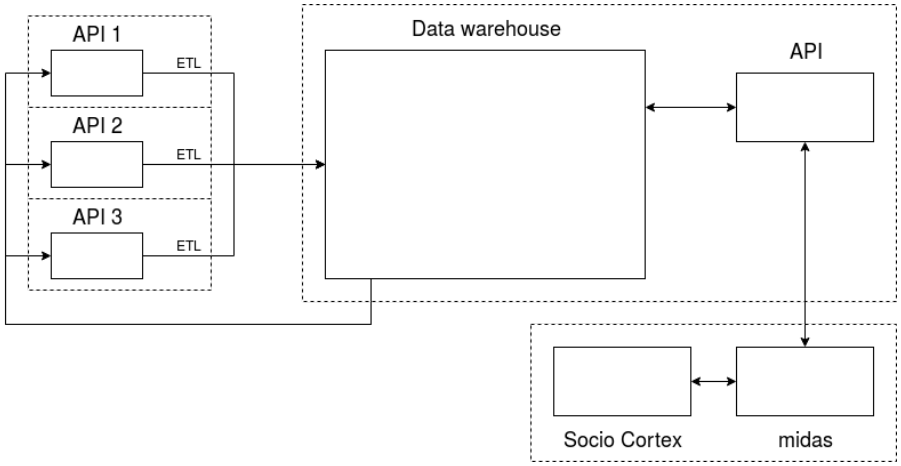

Figure 1.2.: Chair use case architecture.

chair stores it's contacts. *midas*[3] is a project currently under development at the chair as well. It will fetch contact data from SC, send it to the *gordias* API. The enriched data will be sent back to *midas* which will then update the data in *SC*.

In the last part, we will evaluate the tool with data we received from the APIs in part one, mention how this work can be extended in the future and conclude with the results of this elaboration.

1.4.3. Scope of this thesis

Important to mention is that the goal of this thesis is not to gather data from as many sources as possible but rather to create a system with two main focuses:

- Developing a prototype which manages contact data from multiple sources.
- To facilitate the expansion by new sources to the system.

According to this the goal is to create an easily extendable tool to enrich given contact data (more in chapter 8).

APIs as primary data sources

The focus in this work is on APIs for two reasons:

1. Many websites' licences forbid to use crawler and/or scraper on their webpages.
2. It is simply easier to get proper formated data to work with.

[3] `https://www.midas.science`

2. Data sources

2.1. Analysis of requirements and data sources

First of all, we have to find appropriate data sources (websites) to define which information should be used to enrich the contacts. Afterwards, we can search for APIs from different websites and analyze them.

2.1.1. Defining contact requirements

Taking the use case of the *SEBIS* chair of the TUM we define the contact information we are searching for later on. Referring to the section 1.3 these are the requirements:

- Find appropriate contact persons: To know a person's career as well as the areas of expertise is important. We need a collection of keywords describing the person. Any kind of profile description, areas of interest or a list of companies the person worked for in the past would work.
- Invite people to events: Either e-mail or phone number is needed to get in contact with them (getting former is far more likely).
- Improve personal connection on future events: That means we need to know what the people plan to do: Which events they went to in the past, what events they might attend in the future and if they are maybe even hosting own events.

Moreover, it would be useful to have links to different social network profiles of the people to check out additional information about them. According to these requirements we want the following information:

- Name
- E-Mail and/or phone number
- Place of residence
- Interests/Skills
- Education

- Job-related history
- Events (past or future)
- Links to social network profiles

These attributes serve as a starting point for the website search. According to that, on the one hand we need some basic information (name, e-mail, ...) and on the other hand we want to have some more specific, occupational information (areas of expertise, attended events, ...).

2.1.2. Find data sources/websites

In our case, the data sources for the contact information will be social network websites because that is where people mainly disclose information about themselves. The worldwide population was 7.6 billion by the beginning of 2018. There are 4 billion people who use the internet - 3.2 billion use social media actively[17]. According to *globalwebindex*, each internet user has an average of 7 social media accounts [25].

The blockchain ecosystem of the SEBIS chair consists of 27 contacts. For each contact, we know *first name* and *last name* and for 24 contacts we as well know the *company* they are working at at the moment. Searching 27 names individually on Google we looked through the first 20 results for each search. Websites which occured repeatedly and, at the first glance, had an API available we then chose to analyze further:

- **Facebook** is the most popular social media and social network company worldwide. They had over 2 billion monthly active users in the 4th quater of 2017 according to *Statista* [34].
- **Xing** is a career-oriented social networking site for professionals. With over 12 million users it is the largest online business network in Germany, Austria, and Switzerland [18].
- **LinkedIn** is a business- and employment-oriented social networking service. According to *OMnicoreagency* LinkedIn has 500 million users (effective 01.01.2018) [28].
- **Twitter** is an online news and social networking service which had about 330 million monthly users in the 4th quater of 2017 referring to *Statista* [35].
- **Crunchbase** is a leading platform to discover innovative companies, industry trends, investments, and news about myriads of public and private companies globally [8].

- **AngelList** is an online platform for startups, investors and job-seekers [6].
- **F6S** is the world's largest platform for *FounderS* [9].

On the following page is a detailed analysis of who of the 27 persons has a profile on each of the 7 websites.

Name	Company	FB	XI	LI	TW	CB	AL	FS
Alex ***	***	X		X	X	X	X	X
Benjamin ***	***			X				
Bernd ***	***	X	X	X	X	X	X	X
Boris ***	***	X	X	X				
Christian ***	***		X	X	X			
Christoph ***	***		X	X	X			
Christoph ***	***		X	X				
Clemens ***	***		X					
Daniel ***	***		X	X			X	X
Daniel ***								
Dominik ***		X		X	X	X	X	X
Florian ***				X	X		X	X
Florian ***	***	X	X	X	X			
Fritz ***	***	X		X	X			
Joachim ***	***		X	X				
Julia ***	***			X				
Kosta ***	***							
Marco ***	***	X	X	X	X	X	X	X
Marco ***	***		X		X			
Maximilian ***	***	X	X	X	X	X	X	
Omri ***	***	X		X	X			
Paul ***	***		X	X	X	X		
Philipp ***	***				X	X		
Roman ***	***		X	X	X			
Sebastian ***	***		X	X				
Stefan ***	***	X						
Vitus ***	***	X	X	X	X	X	X	X
Total[1]		10	16	21	16	8	8	7

Table 2.1.: Matrix of the chair's contacts and their registration status at each of the seven websites.

Legend to table 2.1: **FB** = Facebook; **XI** = Xing; **LI** = LinkedIn; **TW** = Twitter; **CB** = Crunchbase; **AL** = AngelList; **FS** = F6S.

[1] Effective 20.12.2017

The analysis for people without a company was especially difficult and it was partly not possible to match them. Three persons are represented on all seven websites, two persons on none (or at least they could not be matched). Listing the websites again sorted with most to least (result descending to the represented people of the 27 contacts):

- LinkedIn - 21/27
- Twitter - 16/27
- Xing - 16/27
- Facebook - 10/27
- AngelList - 8/27
- Crunchbase - 8/27
- F6S - 7/27

2.1.3. Selecting appropriate Websites/APIs

Knowing how many of the contacts are on each service we now take a closer look at the website's APIs.

LinkedIn

Via the LinkedIn *Profile API* it is theoretically possible to fetch information from users. The only problem is that we need the user-ID to make that API request [30]. To get the user-ID, using the name and company as search criteria, we have to access the the *People Search API*. In the beginning of 2015 LinkedIn shut down the public access to this API and made it only available for its partnership programs [29]. Furthermore, LinkedIn does not allow to store data, only to cache it up to 24 hours, which would cause performance issues [27].

Assessment: Due to the stated reasons this website cannot be of any use.

Xing

Based on the official documentation the Xing API offers a decent interface. With the *users/find* endpoint we can search for users and their IDs. Just providing the name the API returns a list of possible matches [53]. Taking this ID we could then request heaps of user information using the *user/* endpoint. In reality however, there is no public

API available any more. They only *"focus on a collaboration with a few selected business partners"*. Unfortunatelly we could not find the real announcement, but serveral other references confirm it [16] [52] [54].

Assessment: API is not available any more.

Twitter

With the *users-search* endpoint it is possible to search for users by their name which returns a list of user objects [46]. The rate limit for this endpoint is *900 request / 15 minutes* [47]. Moreover, we can search for tweets. Using the basic (free) version we have access to the tweets of the last 7 days (with premium or enterpriese even more days). Although, we can just get the tweets from the last 7 days that would be enough if we fetch them at least once a week. Rate limit for this endpoint is *1500 request / 15 minutes* [47]. Creating an application in the developer portal and getting an API key is uncomplicated.

Assessment: Multiple endpoints, easy to access and a proper rate limit.

Facebook

"The Graph API presents a simple, consistent view of the Facebook social graph, uniformly representing objects in the graph (e.g., people, photos, events, and pages) and the connections between them (e.g., friend relationships, shared content, and photo tags). Public information can be accessed without a developer key/application key, but is required for private data access." [37]
For the Facebook Graph API four different access tokens exist: *User Access Token (UAT)*, *App Access Toke (AAT)*, *Page Access Token* and *Client Access Token* (used rarely). Creating an application in the developer portal we receive an AAT. Any access to private user profil information is not possible with this token. Therefore, an UAT is needed which is only available once a user signs in at the application via a Facebook login. However, take the UAT of one user another user's information cannot be viewed. Accessing information from public pages, events, etc. is possible with the AAT.
According to the documentation the rate limit for the API is *200 calls per user (of the app) per hour*. Since we just have an AAT we would be restricted to *200 calls / hour* which is little [12]. Another problem is that to be able to get any information about a user via the API we need the user-ID. The API itself does not offer any way to search for users by their name. Pages like `findmyfbid.in` (has an API) offer a way to get the ID but require the person's username instead [13].

Assessment: Even with the user-ID the API barely provides any information about a user just using the AAT. Fetching data from public pages works better, the low rate limits are a problem, though.

AngelList

Unfortunately their API is not available any more. Calling `https://angel.co/api` will result in an *Error 404*[2] and calling `https://angel.co/api/oauth/clients` we get *"API applications are currently closed"* as a response.

Assessment: API is currently closed.

Crunchbase

Access to the full API requires an *Enterprise or Applications License*. *Basic Access Licenses* are limited to the */odm-organizations* and */odm-people* endpoints which return data from the *Open Data Map* [7]. That means we can get basic information, short description as well as URLs from other social media profiles. There is no problem with the licences or data rate limits.

Assessment: We could easily fetch some really useful information from this API.

F6S

On the website they tell that API keys are only issued to enterprise customers [10]. Moreover, they do not allow to make any persistent local copies of any f6s information for longer than twelve hours [11].

Assessment: API cannot be accessed.

2.1.4. Evaluation of Website and API analyse

Summarizing, it can be said that we cannot get as much information automatically from APIs as we can get browsing through the web and looking at user profiles manually. In the most cases we can only get access to user information if the user logs in with his/her own credentials. Hopefully there will be more options to fetch data from social network APIs or other sources in the future.
Implementing data extraction from all APIs mentioned would go beyond the scope of

[2] Effective 02.01.2018 & 05.03.2018

this work (apart from the fact that we can only access a few according to the analysis). However, that is not what we inteded to do anyway. We are just creating a prototype which is easily extendable by new data sources (more in chapter 4 and 8). That is why we chose two APIs to work with for now. Based on the results we decided to implement **Twitter** and **Crunchbase**. Twitter offers a decent amount of information about users as well as a way to search through tweets. There is a total of 1.3 billion accounts, but only 328 million are active [44] [20]. Of those, 44% made an account and left before ever sending a Tweet [33]. With Crunchbase it is easy to access user information as well as information about companies. All eight Crunchbase user (out of the 27 contacts) also have a Twitter profile which makes the interesting case of 'different contact information from differnt sources' likely to evaluate properly. In the next section we will take an even closer look at these two APIs and the data we get from them.

2.2. APIs

2.2.1. Twitter

Using the *python-twitter library*[3] accessing the Twitter API is pretty straight forward. The *GetUsersSearch()*-method from this library searches for users with the name as search query. This accesses the */user/search* endpoint of the Twitter API. From the returned user-object [4] the following attributes may be useful:

- *name*
- *user_name* = Username / User-identifier
- *description* = User description
- *location*
- *url* = User's homepage
- *profile_image_url* = URL to the profile image

To access tweets we use the *GetUserTimeline()*-method with the *user_id* as parameter. This accesses the */statuses/user_timeline* endpoint of the Twitter API and will return a list of tweet-objects[5]. Only the *text* attribute is interesting for us. Based on these texts we try to filter keywords the user is using regularly.

[3] `https://github.com/bear/python-twitter`
[4] `https://developer.twitter.com/en/docs/tweets/data-dictionary/overview/user-object`
[5] `https://developer.twitter.com/en/docs/tweets/data-dictionary/overview/tweet-object`

2.2.2. Crunchbase

No special authentication is requiered. The user key is passed with every request. That is defenitely not the most secure way to communicate, but that is the way the Crunchbase API works. This, however, reduces the implementation effort to a minimum.

As mentioned in the previous section we can only access data from the *Open Data Map (ODM)* with the *Basic Access license*. The ODM inlcudes access to the *people* and *organization* endpoints. The exact contact information returned from the *odm/people* endpoint can be found at `https://data.crunchbase.com/docs/personsummary`. From this attributes the following can be useful:

- *permalink* = Username
- *web_path* = Profil URL without the Crunchbase domain (*person/'permalink'*).
- *first_name*
- *last_name*
- *title* = occupation
- *organization_web_path* = Crunchbase organization URL ending
- *organization_name*
- *profile_image_url* = URL to the profile image
- *homepage_url* = User's homepage
- *facebook_url*
- *twitter_url*
- *linkedin_url*
- *city_name*
- *region_name*
- *country_code* = 3 letter country-code. Many users write the country name, though.

Data fields returned from the *odm/organization* endpoint are described at `https://data.crunchbase.com/docs/organizationsummary`. From those attributes the following can be useful:

- *permalink* = Organization's *"username"*
- *web_path* = Crunchbase organization URL ending (*organization/'permalink'*).
- *name*
- *primary_role* = Role is either *'company'*, *'investor'*, *'group'* or *'school'*.
- *short_description*
- *homepage_url*
- *city_name*
- *region_name*
- *country_code*

2.2.3. Conclusion AIPs

From both, Twitter and Crunchbase, we can get a lot of useful information. We are depending on the users to share the information with us, though.

3. Data warehouse & ETL process

3.1. Data warehousing

Remembering figure 1.1, our goal is to extract data from multiple sources, transform the data, load it into a database and present it to the user via an API. All these steps can be summarized in a more commonly known term: *data warehousing* [40].
Usually this term is connected with the business sector: Information of all areas of an organization is collected to gain better insight into the business and improve the decision-making process. In our case, we do not collect data from different departments of a company but from multiple organizations (online websites). The result will be similar: among others it helps the user to make better decisions (see section 1.3).

3.1.1. Layers of a data warehouse

In general a *data warehouse (DW)* has three layers: (1) data sources, (2) a data staging area (DSA) and (3) the primary data warehouse. The *Extract Transform Load (ETL)* process is linked to these layers and consists of the following steps:

1. *Extract* the data from different data sources.
2. *Transform* and clean the data after it is propagated to the DSA.
3. *Load* it into some sort of database.

Despite the fact that the area of ETL processes is very important, little research has been done in this field. Nevertheless, we give a short overview in the next section.

3.2. ETL process

"ETL development can take up as much as 80% of the development time in a DW project" [31]

The digital world changes everyday: New data sources arise and data gets more diverse. To meet the requirements of this changing environment the ETL processes must be designed for ease of modification. In the following subsections we go in more detail into each step, summarizing the work of El-Sappagh, Hendawi, and El Bastawissy [40].

3.2.1. Extraction

The extraction process consists of two phases: (1) initial extraction and (2) changed data extraction. The second phase only captures changed data since the last extraction using for instance delta technique, system data, audit columns or database log.

3.2.2. Transformation

Out of all three steps of the ETL process, the transformation part is the most expensive one. It includes data cleaning, normalization, transformation and integration [40]. In this stage a series of rules or functions are applied to the extracted data in order to prepare it for loading into the target database. Not everything needs to be transformed, though. That data is known as *'direct move'* data or *'pass through'* data. A metadata repository can be used to store all transformation rules and the resulting schemas.
Different sources have different interfaces and character sets. The challenge is to integrate everything into one system. There are many ways to transform the data. Using standard query language (SQL) to perform the data transformations is a more general approach. It utilizes the possibility of application-specific language extensions, in particular user-defined functions (UDFs). These can be implemented in SQL or a general-purpose programming language with embedded SQL statements and are supported in SQL-99. Language extensions such as the SchemaSQL generically support schema-related transformations. Data transformation can also be accomplished by 'simply' defining an own query language.

An important function of the transformation process is the *cleaning* of data, which aims to pass only proper data to the target.

Data Cleaning

As a basis for detailed clarification the work of Rahm and Do [38] is served. Data cleaning is especially required when integrating heterogeneous data sources and should be addressed together with schema-related data transformations. Other names are *data cleansing* or *data scrubbing*. It deals with detecting and removing errors and inconsistencies from data in order to improve the quality of the data. The ETL process typically has a *wrapper* for each data source to extract the data and a *mediator* for the integration. These systems mostly only provide limited support for data cleaning, though. They are focusing on data transformation for schema translation and schema integration instead.

To understand the concept better we want to introduce a typical data cleaning approach:

1. Data analysis: In order to detect which kinds of errors and inconsistencies exist and need to be removed.
2. Definition of transformation workflow and mapping data: Early data cleaning steps can correct single-source instance problems and prepare the data for integration. Later steps deal with schema/data integration and cleaning multi-source instance problems, e.g. duplicates.
3. The schema-related data transformation steps as well as the cleaning steps should be specified by a declarative query and mapping language as far as possible, to enable automatic generation of the transformation code.
4. Verification: Since some errors only become apparent after applying some transformations multiple iterations of analysis, design and verification steps may be needed
5. Transformation

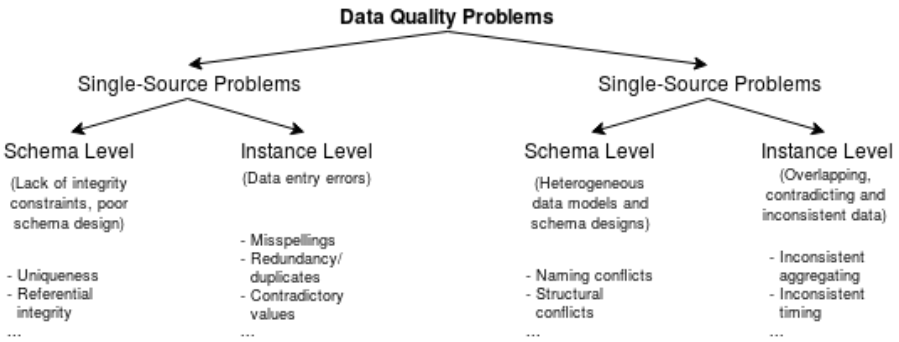

Figure 3.1.: Classification of data quality problems in data sources. This visualization was created following [38].

Figure 3.1 presents an overview of data quality problems. Data quality problems are present in single data collections due to misspellings during data entry, missing information or other invalid data. Multiple sources harbor more advanced problems.

Multi-source problems: The problems from single sources are aggravated when integrating multiple sources. Dirty data may be present in each source and the sources may have a different representation or the information overlaps. This is because the sources

are in general developed, deployed and maintained independently because they have to serve specific needs. A large degree of heterogeneity is the result.

Structural and naming conflicts are the main problems regarding schema design. Structural conflict means if there are different representations of the same object in different sources. They occur in many variations. Naming conflicts arise when either different names are used for the same object (synonyms) or the same name is used for different objects (homonyms).

Overlapping data is another main problem of data cleaning with multiple sources, in particular determining if multiple records refer to the same real-world entity (e.g. a contact). This problem is also referred to as duplicate elimination, the merge/purge problem or the object identity problem. Duplicate information should be removed and complementing information should be merged. The result would be a consistent view of real world entities.

Conflict resolution: In this paragraph we address some problems in more detail and propose general resolutions to the conflicts.

- Extracting values from free-form attributes (attribute split): This concerns name instances and address attributes. Reordering of values within an attribute may be used for word transpositions and value extraction for attribute splitting.
- Validation and correction: This includes spell checking based on dictionary lookup as well as lists with geographic names and zip codes to correct address data.
- Standardization: Attributes values such as date or time should be converted to a consistent format.
- Dealing with multi-source problems: Splitting merging, folding and unfolding of attributes and tables may be needed to achieve schema integration.
- Duplicate elimination: This task is typically performed after most cleaning and transformation steps (further information in [38]).

Cleaning support tools: Most of the tools available for data transformation and data cleaning concentrate on a specific domain, for instance address or name data, or a specific cleaning phase such as duplicate elimination. Due to their domain specification the usage is restricted. To achieve decent results multiple tools must be combined to address all the problems one might encounter creating a data warehouse. Some examples of tools:

- Data analysis and re-engineering tools: e.g. INTEGRITY (Vality)

- Special domain cleaning: e.g. IDCENTRIC (FirstLogic), PUREINTEGRATE (Oracle), QUICKADDRESS (QASSystems)
- Duplicate elimination: e.g. DATACLEANSER (EDD), MERGE/PURGELIBRARY (Sagent/QMSoftware)

So far only sparse research has appeared on data cleaning, although the large number of tools indicate both, the difficulty and importance of the cleaning problem.

3.2.3. Load

Loading the target system is the final step of the ETL process. The extracted and transformed data is written to some sort of database. For large amounts of data additional *data marts* can be created which act as a view for the user to a specific part of the target data warehouse.

3.2.4. ETL models

In this section we present some approaches of how to model ETL processes. One option would be to use *mapping expressions* as Rifaieh and Benharkat do in "Query-based data warehousing tool" [39]. In their approach queries for the mapping between the source and the target data are used to create a data warehousing process. Mapping expression of an attribute is the information needed to recognize how a target attribute is created from the source attribute. Examples of applications where mapping expressions are used are "Generic schema matching with cupid" [32] (mapping expression define the correspondence between matched elements in a database) and "Metadata Management and Data Warehousing" [42] (during the transformation process correspondence between the target- and source-data is defined).

Vassiliadis et al. proposed in "A Framework for the Design of ETL Scenarios" [48] a conceptual model. The models are manipulated by a set of high-level operations, for instance *match*, *union* or *apply function*.

In "Data mapping diagrams for data warehouse design with UML" [31] Luján-Mora, Vassiliadis, and Trujillo introduces a model that is based on the UML (unified modeling language) notations. Both in UML models and in ERD (entity relationship diagram), attributes are embedded in the definition of their comprising 'element' (a class in UML or an entity in the ER) and it is not possible to create a relationship between two attributes. However, sometimes allowing attributes to hold the same roles as entities/classes is exactly what is needed. To achieve that the authors propose the representation of attributes as first-class modeling elements *(FCME)* in UML. Being FCME, classes then act as attribute containers.

The entity mapping diagram (EMD) proposed from El-Sappagh, Hendawi, and El Bastawissy [40] is a relatively new approach. Each non-structured source has its own module called *wrapper*. Wrappers are little tools which automatically extract data from different data sources and convert the received information in some sort of structured format. Tasks of a wrapper are

- fetching data from a remote resource,
- searching for, recognizing and extracting specified data and
- saving this data in a suitable structured format for further manipulation.

Notice that both, data sources and data warehouse schemas, should be defined clearly before starting to draw an EMD.

3.3. Conclusion: Data warehous & ETL process

Our research shows that the main part of building a data warehouse for the contacts will be the ETL process. It should be designed for ease of modification. In order to clean the data we need to analyse the data (see chapter 2) and define a transformation workflow. Data cleansing will include some single source problems as well as some multi-source problems.

Summing up we could say *automate as much as possible while still ensuring data quality*. Now we will apply the knowledge of the first chapters to build the prototype.

Part II.

Main part

4. Software architecture

First of all, we decided to go with Python (Python 3.6) because it is a simple but powerful programming language. In addition, the expansive library of open source data analysis tools, web frameworks, and testing instruments make its ecosystem one of the largest out of any programming community [51].

4.1. Database

We are using PostgreSQL because it is the recommended database for working with Python[1] and it satisfies our needs.

4.1.1. Tables

The database consists of four tables: two tables for the merged contact data and two tables for the sources' data. The schema has two sorts of tables: (1) main tables and (2) value tables. The main tables only hold metadata. The values tables hold the actual contact information of the corresponding main table. Figure 4.1 shows the *entity-relationship diagram* (ER diagram).

4.1.2. Metadata

In addition to the contact data fields we need to store some more information with each contact in the database.

- *contact_id*: 8-bit unique contact-ID
- *last_sync*: Timestamp of the last time this contact was synchronized (value in seconds). This information is only stored in the main *contacts* table, not in the source tables.
- *sync_interval*: If this time has passed since the last synchronization the contact needs to be synchronized again (value in seconds).

[1] `https://www.fullstackpython.com/databases.html`

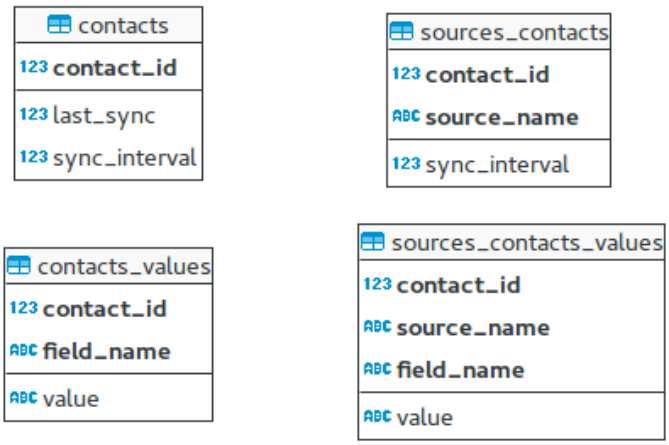

Figure 4.1.: ER diagram of the database.

4.2. API

Using Python, a suitable framework for creating REST-API is *bottle*[2]. Our API has the following endpoints:

- */contact*: Enriches a single contact (POST-request). Parameters are first name, last name and organization.
- */contacts*: Enriches multiple contacts (POST-request). Parameter is a string in JSON format containing a list of basic contact attributes: for each contact which should be enriched we need first name, last name and organization.

First name and last name *must* be provided in every case. If organization data is missing matching of the correct contact at the individual data sources cannot be guaranteed. Both endpoints return a JSON object with the found contact attributes.
An additional parameter *age* can be added to each request. Providing this information the user can define how old the enriched data can be. If the last synchronization time of a contact lies further back than the time of the request subtracted by the age of the data will be synchronized with the sources. The value of this parameter is specified in hours.

Another endpoint is just used for internal synchronization purpose (of which more in chapter 6).

[2]Bottle Github link: `https://github.com/bottlepy/bottle`

4.3. Data sources

As already mentioned in chapter 1 the prototype should be modular. In other words, it should be easy to add new data sources to the system.

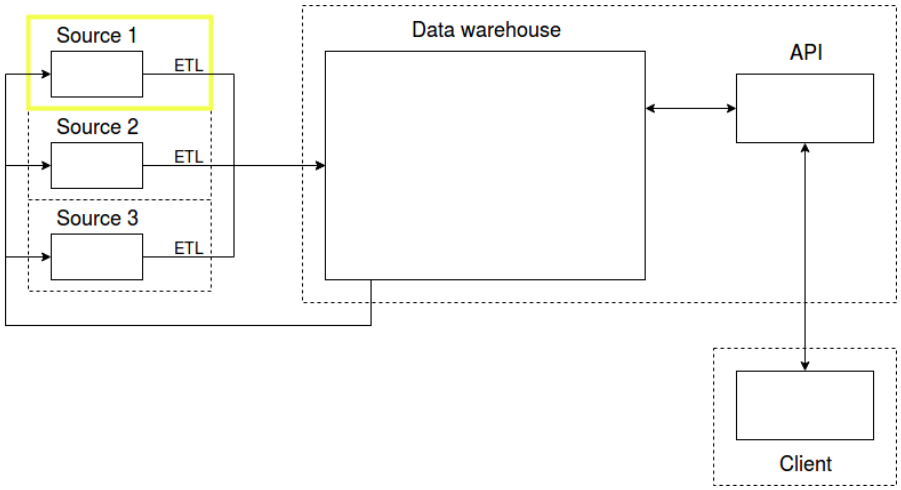

Figure 4.2.: General architecture with one module highlighted.

The highlighted section in figure 4.2 is one module. A module consists of two parts:

- Some logic to fetch the data from the source (usually an API).
- An adapter which transforms the data.

That means one module is an ETL process in itself: It extracts data from the actual source, transforms it and passes it to the main logic where it gets loaded into sources table in the database.

4.3.1. Module architecture

Everything concerning data sources is in the *sources* folder of the project. As figure 4.3 shows the folder contains 3 things:

- A configuration file for the sources (sources_config.json)
- A file which includes an abstract source class (source_class.py)
- Source module folders (in figure 4.3 *tw* and *cb*)

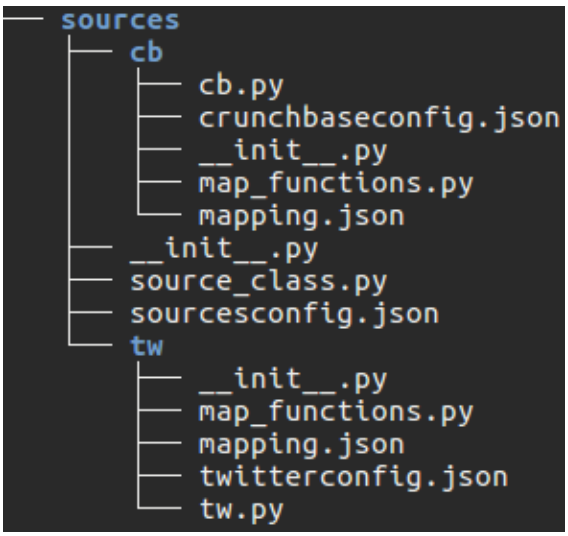

Figure 4.3.: Exemplary path tree of the *sources* folder.

Source's configuration file

Every module has to have an entry in this file so the system knows that it exists.

```
{
   "name" :"NAME",
   "path" :"abc/xyz",
   "class_name" :"CLASSNAME"
}
```

This snippet is an exemplary entry of the configuration file. It has 3 attributes:

- *name*: Simply the name of the source (e.g. "Twitter").
- *path*: Path to the file that contains the class *class_name* (e.g. "tw/tw").
- *class_name*: Name of the class that inherits from the abstract source class (e.g. "Twitter").

Abstract source class

In order to ensure a modular architecture we need a function the *Datacollector* can call for each module to get the source's data. Therefore, we implemented an abstract source class *Source* which contains an abstract function called *get_data()*. The classes specified by the *class_name* attribute in every entry in the *sources_config.json* file have to implement this method.

Source module folders

The source module folder has to contain a class that inherits from the abstract *Source* class and implements the *get_data()* function. Moreover, we need an *__init__.py* file (it can be empty) and a mapping.json file which defines the mapping of the source's data fields to the internal data structure (more in chapter 5). Additionally, it can contain a file called *map_functions.py* which contains source-specific mapping functions (more in chapter 5).

4.3.2. Adding a new module

A new source can be added to the system in 6 steps:

1. Create a new folder in the *sources* directory.
2. Create a new python file in that folder.
3. Create a class which inherits from the class defined in the *source_class.py* file and implement the *get_data()* function.
4. Add a new entry to the *sources_config.json* file.
5. Adding a mapping configuration file *mapping.json*.
6. If needed add an additional file *map_functions.py* with source-specific map functions.

4.4. Enricher & Datacollector

The sequence diagram in figure 4.4 shows the sequence of how the different parts of the prototype work together. First the Enricher checks if the contact is already in the database. If the contact does not exist the *collect_data()* function of the Datacollector is called. The Datacollector then calls the *get_data()* function of each source module, maps the data to the internal data structure and returns a dictionary of all sources and their mapped data fields. The Enricher takes the returned value and merges all sources' data into one contact. After that all sources' data as well as the enriched contact data is saved to the database.

However, if the contact already exists in the database the next steps depend on the *age* parameter of the request. If the parameter indicates that the stored data is too old the *collect_data()* function is called. Otherwise, the contact is just returned.

Figure 4.5 shows the process in more detail. If a list of contacts should be enriched the just described process is gone through for every contact.

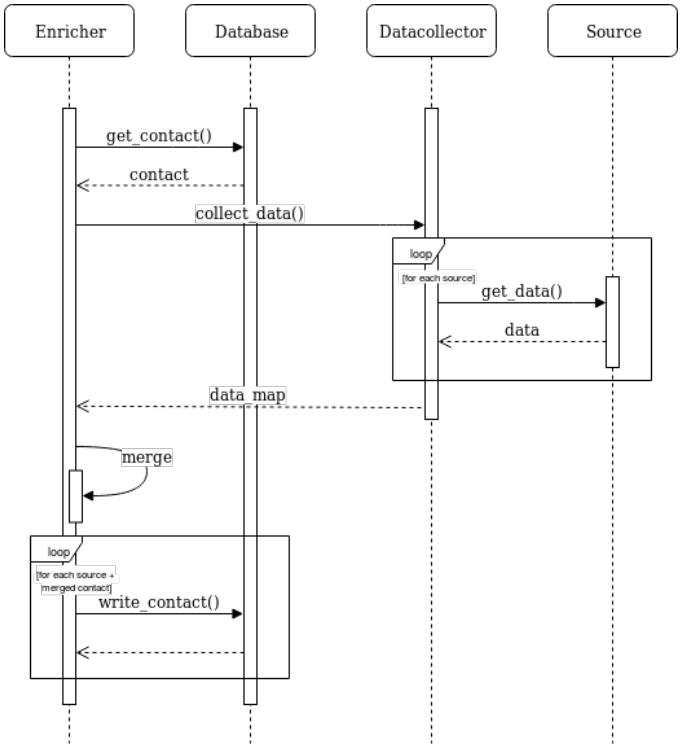

Figure 4.4.: Sequence diagram starting at the Enricher.

4.4.1. Contact matching

Matching of the 'right' contact happens when a source module requests data from the actual source (API). Source APIs may return a list of contacts 'matching' the given request. Every source is different concering the API. When choosing a contact from a list of contacts we have to keep in mind that it has to be possible to match this contact, once it is in the database, with the data obtained from a request (first name, last name, organization).
To facilitate the matching process we implemented some functions: phrase similarity calculation and others (more in the project's README file[3]).

In the next chapter we explain the mapping and merging process in more detail.

[3] *goridas* Github link: `https://github.com/yemboo2/gordias`

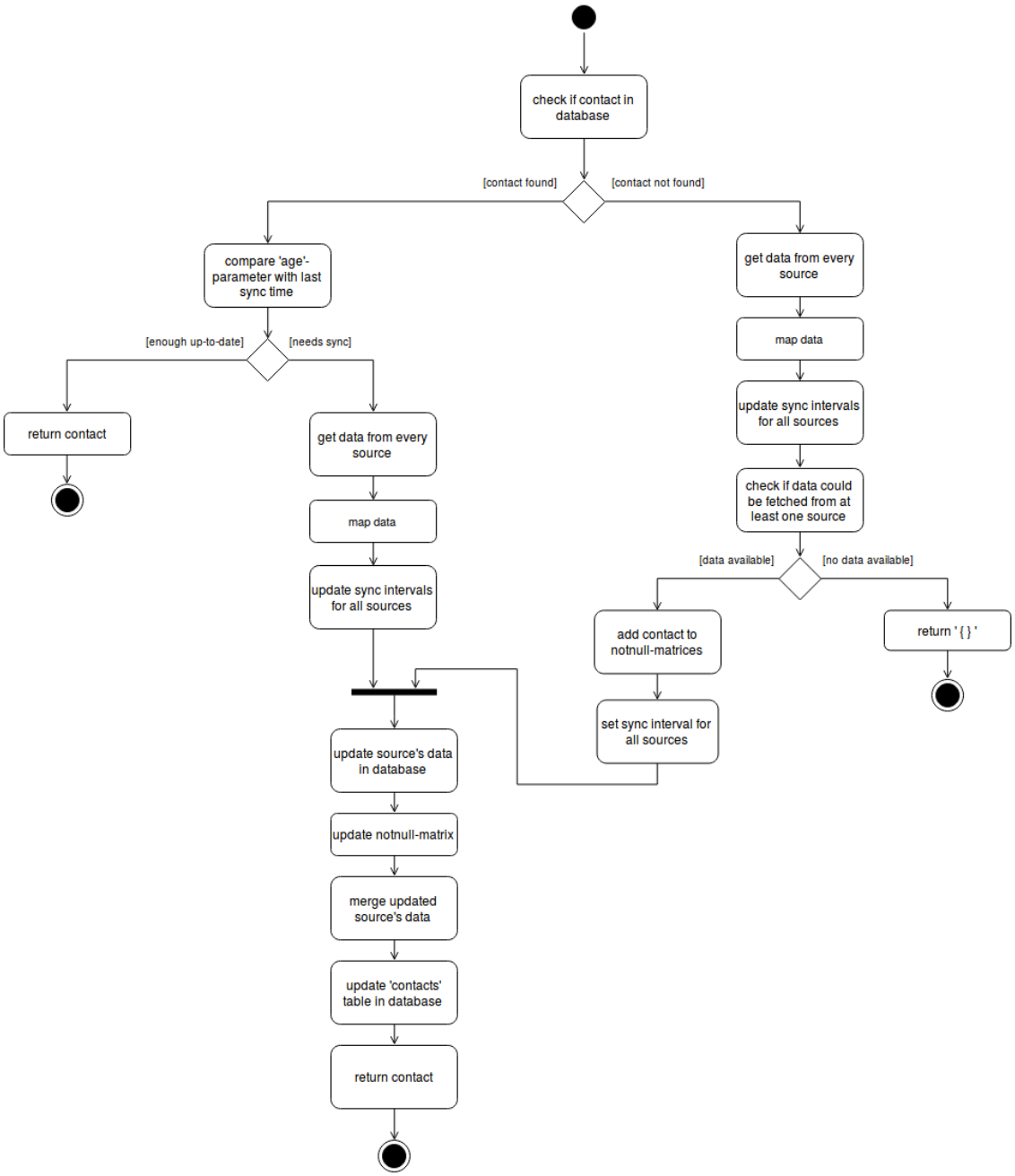

Figure 4.5.: Enrichment process in detail.

5. Mapping & Merging

5.1. Mapping

A mapping expression of an attribute is the information needed to recognize how a target attribute is created from the source attribute. In section 3.3 we concluded that data transformation should be specified by a declarative query and mapping language as far as possible to enable automatic generation of the transformation code.

The Datacollector calls the *get_data()* function for every module class. This function has to return a key-value-map (called *dictionary* in Python). The key is the field name, e.g. firstname, and the value is the according field value, e.g. "Alex". To map that data to our internal data structure we defined our own query language. The query language is written in JSON syntax and looks like the following:

```
{
        "identity" :[
                {
                        "source" :"url",
                        "target" :"homepage"
                },
                {
                        "source" :"profile_image_url",
                        "target" :"profile_image_urls"
                }
        ],
        "extract_keywords" :[
                ...
}
```

This snippet of a mapping file has a set of map-function-names at the first level. Each map-function-name again has a list of JSON objects as value. These objects have a *source* and a *target* field. The value of the *source* attribtue defines the field name in the source data object and the value of the target attribute defines the field name of the data object it should be mapped to.

Reading a configuration file like the above the Datacollector takes the values of the source field names, applies the mapping function and saves it to the according target fields. If two entries with the same target-field-name exist in a source's mapping file the values will be merged with the appropriate merge function of the target field.

5.1.1. Map strategies

To get the respective map function the datacollector first checks the general *map_functions.py* file and if it cannot be found a source specific *map_functions.py* file is checked (if exists). In the main *map_functions.py* file the following map strategies exist:

- **Identity**: The value just gets returned.
- **Country code**: 3-letter country code is translate to a country name using the *pycountry-convert* library[1].
- **Name split**: A String containing the full name is splitted and first and last name are extracted using the *HumanName Parser* library[2].
- **Location split**: An unstructured location string gets splitted and the country and city values are extracted using the *GeoText* library[3].
- **Keyword extraction**: For the keyword extraction we use the *natural language toolkit (NLTK)* [3]. First we tokenize the text. Afterwards, part-of-speech tagging [43] is used to tag each token. Then we filter the tokens on their tags and extract the nouns. Having a list of nouns we then create a set and count the occurrences for each noun. Taking the minimum and maximum count a threshold is calculated and all tokens occurring more often than that threshold are returned as keywords.

5.2. Merging

The Datacollector returns a map of sources and their data objects. These data objects have to be merge into one contact data object. Every data object has the same attributes - some may be empty if the source could not provide useful information. Not every field is merged the same way. Different fields require different merge strategies. In the *field_merge_mapping.json* file every field is assigned a merge strategy (merge function):

[1]Github link: `https://github.com/TuneLab/pycountry-convert`
[2]Github link: `https://github.com/derek73/python-nameparser`
[3]Github link: `https://github.com/elyase/geotext`

```
{
        ...
        "email" :"collect_emails",
        "city" :"best_field",
        ...
}
```

Most currently existing merge strategies are pretty straight forward. The *'best field'* strategy is a bit more complicated, though.

```
def best_field(contact_id, current_value, new_source_value_map):
  weighted_field_count_dict = dict()
  for source, value in new_source_value_map.items():
    weight = notnull.get_nn_score(contact_id, source)
    if value in weighted_field_count_dict:
      weighted_field_count_dict[value] += weight
    weighted_field_count_dict[value] = weight

  if not weighted_field_count_dict:
    return ""

  return max(weighted_field_count_dict, key = weighted_field_count_dict.get)
```

Here we do not care about the current merged value, just the current values retrieved from the sources (which are in the variable *new_source_value_map*). *weighted_field_count_dict* is a dictionary with the field values as keys and their weight as value. For every value in the dictionary *new_source_value_map* we calculate the weight (more about that in the next section). If the value is already in the dicitionary (as a key) we add the weight to the existing weight. Otherwise, we add this value and the weight to the dictionary. In the end we return the key, which actually is the value for the field, with the maximum weight.

5.2.1. How can the weighting of a source's information be assessed?

We generate a weight matrix $SOURCES \times FIELDS$. Weights in this matrix get updated/changed as new contacts get added or old ones synchronized.

Metric(s)

To determine the weight of a contact for a specific source we need some metric(s). At first two metrics seem reasonable:

1. How many not null fields does a contact from a specific source have.
2. How often does a field match the respective field in the merged contact table.

Metric 1: How many not null fields does a person in relation to the other persons for this source and in relation to information of other sources for this person have. The more information there is about a person of a specific source, the more it seems the owner of this profile takes care of this account. That means the information is more likely to be correct.

Metric 2: How often does a field match the respective field in the merged contact table. The relations would be the same as with metric 1. However, this approach is not very reasonable because we cannot say for sure if the merged data is correct. That implies that once the merged value is incorrect the error will propagate. Unless we do not have some sort of training set this metric should not be used (see chapter 8).

Mathematical foundation

Let S be the set of sources, C be the set of contacts, F be the set of fields (euqivalent with the fields returned from the API) and $F(s) \subseteq F, \forall s \in S$ be the set of fields which one source actually contributes to (set of all target field names of the mapping for the source).

$$\forall s \in S, \forall c \in C: \quad w(c,s) = \frac{avgc(s)}{avgs(c)}$$

$$avgs(c) = \frac{avgsumnns(c)}{|S|}$$

$$avgc(s) = \frac{avgsumnnc(s)}{|C|}$$

$$avgsumnns(c) = \sum_{s \in S} sumnn(c,s)$$

$$avgsumnnc(s) = \sum_{c \in C} sumnn(c,s)$$

$$avgnn(c,s) = \frac{\sum_{f \in F(s)} nn(c,s,f)}{|F(s)|}$$

$$nn(c,s,f) = \begin{cases} 0, \text{ if } f \text{ null} \\ 1, \text{ if } f \text{ not null} \end{cases}$$

The function nn returns 0 if the field f for contact c of source s is null or 1 if it is not null (null is equivalent to empty in this case). In the $avgnn(c, s)$ function we first calculate the sum of all not null fields for this source contact. This value could be used in $sumavgnnc(s)$ without consequences as we only build the sum over all contacts' $avgnn(c, s)$ values (source stays the same). However, in the function $sumavgnns(c)$ we build the sum over all sources in S. Because it may be that $\exists s1 \in S, \exists s2 \in S, s1 \neq s2 : F(s1) \neq F(s2)$ building the average of all sums would lead to an incorrect value[4]. Instead, we already build the average of the sums of the not null fields. At the first glance $w(c, s)$ seems not very intuitive: why setting two averages in relation to one another? It is, however, already the transformed formular.

$$w(c,s) = \frac{\frac{avgnn(c,s)}{avgs(c)}}{\frac{avgnn(c,s)}{avgc(s)}} = \frac{avgc(s)}{avgs(c)}$$

The reason why we set two relative values in relation to one another is the following: Calculating the sum of not null values of one contact for a source and set it in relation to the average over all contacts for this sources it tells us if the user disclosed more or less information in comparison to other users. Assuming a user enters less information at this source, the average the weight would be small. However, if the user in general discloses little information on all his/her social media/network profiles the weight for the contact of this source should not be small.

$w(c, s)$ is the value used to calculate the weight in the *best_field* merge strategy.

[4]There may be 16 fields overall but the source's data is only mapped to 8.

6. Synchronization

When referring to *synchronization* in this chapter we do not mean the process of updating a contact's information if it is requested via the API. This happens when a contact already exists in the database but the *age* parameter of the request indicates that the last time the contact's data was checked against the sources lies to far back in time. We mean the separate process of synchronizing contacts regularly independent of API requests from the framework users.

6.1. Why data synchronization?

6.1.1. Response time

Theoretically we could fetch all the data from the sources and return it to the user every time we receive a request. However, imagine a user sends a request with a list of 50 contacts. Assuming the system has 3 sources with each source performing 1 API call this would result in 150 API calls. Presumably the response time for 1 API call is at least 100ms (and that is an optimistic assumption), this would lead to a response time of at least 15 seconds (not including the computation time of our prototype itself). Despite the fact, that it is inevitable to wait longer when contacts are enriched for the first time, response times for request with contacts which have already been enriched before can be reduced significantly if stored in a datawarehouse and synchronized consistently.

6.1.2. Data aging and its impact on data quality

The negative impact of data aging on the data quality is the reason why just storing the data in the database is not enough. If contact data is not synchronized regularly data quality worsens. The more time passes the more likely it is that current source data does not match the data in our database [49].

The rest of this chapter is divided into two parts: (1) defining how often to synchronize contact data and (2) how our synchronization process looks like.

6.2. Synchronization interval

First of all we need to clarify what data we actually fetch from the sources and how likely it is that this information will change.

6.2.1. Initial synchronization interval

If a contact gets added to the database its *sync_interval* attribute will be the average of all other contacts' sync intervals of the respective source. In the beginning there are no users in the database which means we need to predefine an initial sync-interval value. As a result of not having any studies about how often users change/update their social media profiles it is difficult to predefine an appropriate synchronization time for the first user in the database. The next best approach is to look at studies and surveys covering how often variables like jobs or e-mail addresses changes for people in reality:

Name: We do not take this into consideration because it normally only changes when someone marries.

E-mail: On average 2% of one's e-mail list gets outdated every month because people change e-mail addresses, jobs, organizations, etc. That means in average persons change their e-mail every 4 years [1]. Another survey says that 30% of subscribers change e-mail addresses annually and 17% of Americans create a new e-mail address every 6 months [50].

Location: In Great Britain people move home every 23 years on an average [22]. Every fourth American moves every five years according to a Gallup Survey in 2013 [2]. The average American will move 11.4 times in its life, referring to [19].

Job: According to *The Balance* people change their jobs between ten to fifteen times during their whole career [21]. EdSurge, a LinkedIn executive, said in an interview that people change jobs fifteen times over their lifetimes [2].

Organization: Once a person changes a job also the related organization data in our database changes.

Social media/network profile URLs: These hardly ever change. Therefore, we will not take them into consideration.

Profile images: They vary from platform to platform: Facebook (which owns Instagram) users change their profile pictures about every 5 months, LinkedIn users every 2.1 years and Twitter users about every 1.8 years. In average people do change their profile pictures every 2 years [41].

Keywords (interests, etc.): Currently keywords are extracted from profile descriptions and Tweets. We could not find anything related to profile description. However, we can retrieve Tweets from the last 7 days.

Taking all these facts into account (especially the fact about Tweets) synchronization should take place at least once a week.

6.2.2. Updating the synchronization interval

Metric(s)

To update the synchronization interval we need some metric(s). At first two metrics seem reasonable:

- How often does a source's data change? Often -> reduce sync interval; rarely -> increase sync interval. The value from this metric could be used for all contacts of a source. However, contacts are synchronized individually so every contact needs to have its own value for this metric.
- How often do people request contacts via our API? The more often a contact is requested the more it should be ensured that it is up-to-date (because the more it is actually used). A contact requested every once a year does not need to be synced every week.

If, however, the second metric is used and a contact is not requested for a long time this could lead to a pretty high sync interval, even though data may change. A solution would be to weight the value of the 'has data changed' metric more than the value of the 'how often is a contact requested' metric. However, it would still influence the interval value. The less we want it to influence the sync interval the less we have to weight it. Keeping in mind that we want to keep the data up-to-date even if it is not requested very often, because it is important to get this information to secure a correct and updated nn_matrix (see chapter 5), this metric should be weighted even less. The less we weight it the less justified is its existence in the first place. The goal is to make synchronization as efficient as possible while still keeping the data up-to-date.

Taking everything in consideration it seems reasonable to drop the second metric and let the sync interval only be defined by how often attributes from a source change.

Calculation

Every time a contact gets synchronized and the data is fetched from the sources we check if the data from each source matches the data in the according source table in the database. If the fetched data matches the data from the database the sync interval will be increased, if it does not match ($\hat{=}$ data changed) the sync interval will be decreased:

- Match: $new\_sync\_interval = old\_sync\_interval * 1.05$
- No match: $new\_sync\_interval = old\_sync\_interval * 0.95$

Every source table obviously has its own sync-interval attribute. Theoretically the information of a source for a contact could be synchronized separately. However, every time the information of one source for a contact is fetched and the data changed it has to be merged. Additionally, if the last synchronization time of one source for a contact is too old, once a contact is requested over the API with an *age* parameter, all sources will be synchronized again. As a result, it is reasonable to take the minimum of all sources' sync intervals and make this minimum the decisive value for the synchronization process. It then is saved to the *contacts* table (the table with the merged contact information) of the database.

6.3. Synchronization process

Based on the information of the last synchronization time and the synchronization interval we can calculate the date when a contact should be synchronized again:

$$next\_sync = last\_sync + sync\_interval$$

Using this formula we can create a sync table, looking like the exemplary table 6.1.

contact_id	**next_sync**
12345678	1523059200
87654321	1523060789
...	...

Table 6.1.: Exemplary synchronization table

As we can see in figure 6.1 the program has two threads: One thread handles the API requests and the other thread does the regular synchronization.

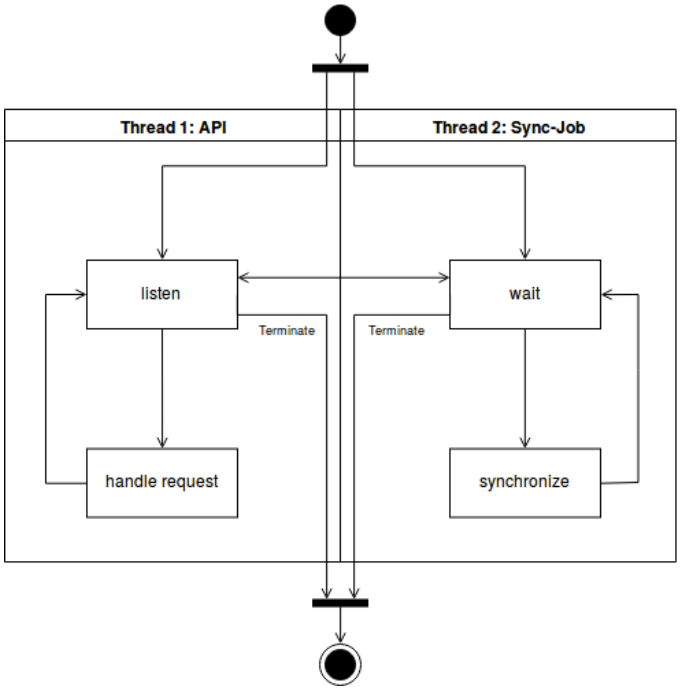

Figure 6.1.: Abstract activity diagram of the prototype.

The sync-job loop has 4 steps:

1. Update the sync table.
2. Get the smallest *next_sync* value (of the table).
3. If the value is smaller than or equal to zero the contact data will be updated.
4. If the value is greater than zero we wait for '*next_snyc*' seconds.

In step 3 we create a request for the program's API. Doing this we implement some sort of lock on the enriching functions of the prototype because the API requests are processed successively. As a result a 'change' of the thread can either happen in the *listen* or in the *wait* state. Contacts are synchronized one after another. Even if, according to the sync table, more contacts need to be updated we return to the *wait* state before selecting the next contact of the sync table. This gives the program the chance to change to the other thread and handle any API requests occurred in the meantime.

This chapter complements the prototype.

Part III.

Results

7. Evaluation

7.1. Clarifications

First of all we need to clarify two points: the *source data dependency* and the impact of the *matching decisions*.

7.1.1. Source data dependency

A straight forward evaluation is not possible because we cannot just judge the system based on the fact with how much information each contact is enriched. We depend on the information the user enters at the different social network websites. If all data fetched from a source is incorrect or the user just does not share any information on any website, it cannot be expected to retrieve fully and correctly enriched contacts.

7.1.2. Matching decision

Another role plays the matching of the right user if we retrieve multiple users for one name (from a specific source). If a contact might be the wrong one and the alternative would be no contact, a decision has to be made: Taking the contact and risk having maybe wrong data in the system or taking no contact and risk maybe losing correct data.

7.2. Contact evaluation

We used the contacts of table 2.1 for the evaluation. 27 contacts, 16 of them could be manually found on Twitter, 8 on Crunchbase (of which all have a Twitter account, saying they are included in the 16 contacts found on Twitter). 3 contacts had no organization information.

The (first) evaluation step is to check how many contacts could actually be matched with the source contacts. Table 7.1 shows the evaluation results. 7 of the 16 Twitter contacts and 5 of the 8 Crunchbase contacts could be matched. A reason for the relative low match rate with Twitter is that no 'organization'-like attribute exists in Twitter profiles. That means the match (apart from name) depends on the description. Another reason is

Name	Twitter	Crunchbase		TW	CB
Alex ***	X	X			X
Bernd ***	X	X		X	
Christian ***	X				
Christoph ***	X			X	
Dominik ***	X	X		X	
Florian ***	X				
Florian ***	X			X	
Fritz ***	X				
Marco ***	X	X		X	X
Marco ***	X				
Maximilian ***	X	X		X	X
Omri ***	X				
Paul ***	X	X			X
Philipp ***	X	X		X	X
Roman ***	X				
Vitus ***	X	X			
Total	16	8		7	5

Table 7.1.: Matrix of the chair's blockchain ecosystem related contacts. The columns *Twitter* and *Crunchbase* indicate if the people are actually registered at that website. *TW* and *CB* indicate if the contacts could actually be matched.

the matching method mentioned in section 7.1.2. For the sources we implemented so far we decided that it is more important to have maybe less but correct data. 5 contacts could be matched fully, 4 contacts could be partly matched and 7 contacts could not be matched at all.

The next evaluation step would be to check if the merged values are values a human would choose, having all the information of the different sources. However, only three persons which are present at both sources could be matched fully (on both websites). We cannot conduct an expressiv evaluation with only three contacts. As a result, we have to postpone this step until we have more contacts to work with (see chapter 8).

8. Further work

If the test phase shows that the extracted keywords are not very appropriate the keyword extract function should be improved through adjusting the tag filter and/or the word prioritization process. It could as well be extended by not only extracting single keywords but also phrases using for instance *rake-nltk*[1]. Depending on the future use cases the API may want to be extended to also handle different files, e.g. spreadsheet files containing the contacts instead of Strings in JSON format. Hardly any sources provide an e-mail address. People typically follow specific rules when generating/choosing an e-mail address [23]. We could try to generate e-mail addresses by using common patterns and then use some e-mail verification service like *Hunter*[2] to validate them. This process could be added as a post-action after the merging.

In chapter 5 we concluded that the metric (How often does a attribute match the respective attribute in the merged contact table) is not reasonable. However, if we could get a proper training set this metric can be used as well.

We explained the synchronization process but not how it potentially could influence the quality of sources (as a third metric). If a contact of a source changes quite often this could mean two things: (1) the person cares about its account and keeps the data up-to-date or/and (2) the information is not reliable because the information changes more often than people change in reality.

As mentioned in chapter 7 another evaluation step could be done: Checking if the merged values are values a human would choose having all the information of the different sources. To properly conduct this step more contacts and sources are needed.

We want to empathize that the created prototype is a framework and the two added sources should just exemplary show what is possible. That means the clear main goal in the future is to add new source modules to the framework to further increase the amount of information gathered and as a result imporve data quality. In a final step big companies like Facebook, Xing, LinkedIn could be persuaded to offer us their user data (of course not private information but public available information). We then use it, further expand the framework and improve the enriching process. Those companies then could access the tool and benefit themselves.

[1]Github link: `https://github.com/csurfer/rake-nltk/blob/master/rake_nltk/rake.py`
[2]Offers API access: `https://hunter.io/api/docs`

9. Conclusion

In this thesis we built a prototype called *gordias* which takes the information of several APIs, transforms it and stores it in a data warehouse. The gathered data can then be requested over an API. The goal was to create a tool which manages contact data from multiple sources and facilitates the expansion by new sources to the system. In order to do that we first analyzed multiple social media/network websites and their APIs. The result was, that we cannot get as much data as we expected at first. Either they close their API or make public information of users only accessible if the user logs in with his/her own credentials. We exemplary implemented data extraction from two sources: Twitter and Crunchbase. Research on the *data warehouse* and the *ETL process* area was done to provide the fundamental knowledge needed to build a prototype. An easy modifiable and modular structure was developed to facilitate the adding of new sources to the system. For data mapping and merging purposes we designed our own query languages. The creation of appropriate metrics enabled us to asses the correctness of data and as a consequence improve data quality. An adapting synchronization job with adapting synchronization interval keeps the data in the data warehouse up-to-date. Evaluation showed that the matching of the right contacts is more complicated than we thought. About half of the users' profiles could only be matched manually. For the next evaluation steps we need more contacts (& sources).

Multiple iterations of the analysis, design and verification steps will for sure further improve the system. We are depending on the users to share the information with us, though. Summing up, the created framework serves its purpose of enriching contacts with a modular source management system but even more contact information is desirable.

List of Figures

List of Tables

Bibliography

[1] *15 Email Statistics That Are Shaping the Future*. http://www.convinceandconvert.com/convince-convert/15-email-statistics-that-are-shaping-the-future/. Accessed: 03.04.2018.

[2] *381 Million Adults Worldwide Migrate Within Countries*. http://news.gallup.com/poll/162488/381-million-adults-worldwide-migrate-within-countries.aspx. Accessed: 03.04.2018.

[3] Steven Bird and Edward Loper. "NLTK: the natural language toolkit." In: *Proceedings of the ACL-02 Workshop on Effective tools and methodologies for teaching natural language processing and computational linguistics* 1 (2002), pp. 63–70.

[4] Sudarshan Chawathe et al. "The TSIMMIS project: Integration of heterogenous information sources." In: (1994).

[5] *Clearbit*. https://clearbit.com/enrichment. Accessed: 22.03.2018.

[6] *Crunchbase - AngelList*. https://www.crunchbase.com/organization/angellist. Accessed: 06.03.2018.

[7] *Crunchbase API Docs*. https://data.crunchbase.com/docs. Accessed: 06.03.2018.

[8] *Crunchbase - Crunchbase*. https://www.crunchbase.com/organization/crunchbase. Accessed: 06.03.2018.

[9] *Crunchbase - F6S*. https://www.crunchbase.com/organization/f6s. Accessed: 06.03.2018.

[10] *F6S - API Docs*. https://www.f6s.com/developers/apis/profile. Accessed: 06.03.2018.

[11] *F6S - Privacy Policy*. https://www.f6s.com/privacy-policy. Accessed: 06.03.2018.

[12] *Facbook - Rate Limiting on the Graph API*. https://developers.facebook.com/docs/graph-api/advanced/rate-limiting. Accessed: 07.03.2018.

[13] *Find Your Facebook ID*. https://findmyfbid.in. Accessed: 07.03.2018.

[14] *Freshsales*. https://support.freshsales.io/support/solutions/articles/217625-does-freshsales-enrich-lead-contact-and-account-information-. Accessed: 22.03.2018.

[15] *Fullcontact Homepage*. https://www.fullcontact.com. Accessed: 06.03.2018.

[16] *Github - Xing consumer key and secret*. https://github.com/xing/xing_api/issues/14. Accessed: 06.03.2018.

[17] *Global social media research summary 2018*. https://www.smartinsights.com/social-media-marketing/social-media-strategy/new-global-social-media-research/. Accessed: 30.03.2018.

[18] *Google Playstore - Xing*. https://play.google.com/store/apps/details?id=com.xing.android. Accessed: 06.03.2018.

[19] *How Many Times Does The Average Person Move?* https://fivethirtyeight.com/features/how-many-times-the-average-person-moves/. Accessed: 03.04.2018.

[20] *How Many Users Does Twitter Have?* https://www.fool.com/investing/2017/04/27/how-many-users-does-twitter-have.aspx. Accessed: 30.03.2018.

[21] *How Often Do People Change Jobs?* https://www.thebalance.com/how-often-do-people-change-jobs-2060467. Accessed: 03.04.2018.

[22] *How often the average Brit moves house*. http://www.idealhome.co.uk/news/zoopla-average-brit-moves-home-181281. Accessed: 03.04.2018.

[23] *How to Guess a Surprising Number of Email Addresses*. https://www.lifewire.com/how-to-guess-email-addresses-1170885. Accessed: 05.04.2018.

[24] William H Inmon. *Building the data warehouse*. John wiley & sons, 2005.

[25] *Internet Users Have Average of 7 Social Accounts*. https://blog.globalwebindex.net/chart-of-the-day/internet-users-have-average-of-7-social-accounts/. Accessed: 30.03.2018.

[26] *iSEEit*. https://now.iseeit.com/. Accessed: 22.03.2018.

[27] *LinkedIn - API Terms of Use*. https://developer.linkedin.com/legal/api-terms-of-use. Accessed: 06.03.2018.

[28] *Linkedin by the Numbers*. https://www.omnicoreagency.com/linkedin-statistics/. Accessed: 06.03.2018.

[29] *LinkedIn - Developer Program Transition Guide*. https://developer.linkedin.com/support/developer-program-transition. Accessed: 06.03.2018.

[30] *Linkedin - Profile API*. https://developer.linkedin.com/docs/guide/v2/people/profile-api. Accessed: 06.03.2018.

[31] Sergio Luján-Mora, Panos Vassiliadis, and Juan Trujillo. "Data mapping diagrams for data warehouse design with UML." In: *International Conference on Conceptual Modeling* (2004), pp. 191–204.

[32] Jayant Madhavan, Philip A Bernstein, and Erhard Rahm. "Generic schema matching with cupid." In: *vldb* 1 (2001), pp. 49–58.

[33] *Metrics suggest 44% of Twitter uses never tweet*. https://www.slashgear.com/metrics-suggest-44-of-twitter-uses-never-tweet-14325098/. Accessed: 30.03.2018.

[34] *Number of monthly active Facebook users worldwide as of 4th quarter 2017 (in millions)*. https://www.statista.com/statistics/264810/number-of-monthly-active-facebook-users-worldwide/. Accessed: 06.03.2018.

[35] *Number of monthly active Twitter users worldwide from 1st quarter 2010 to 4th quarter 2017 (in millions)*. https://www.statista.com/statistics/282087/number-of-monthly-active-twitter-users/. Accessed: 06.03.2018.

[36] *Pipl*. https://pipl.com/. Accessed: 22.03.2018.

[37] *ProgrammableWeb - Facebook Graph API*. https://www.programmableweb.com/api/facebook-graph. Accessed: 06.03.2018.

[38] Erhard Rahm and Hong Hai Do. "Data cleaning: Problems and current approaches." In: *IEEE Data Eng. Bull.* 23.4 (2000), pp. 3–13.

[39] Rami Rifaieh and Nabila Aïcha Benharkat. "Query-based data warehousing tool." In: *Proceedings of the 5th ACM international workshop on Data Warehousing and OLAP* (2002), pp. 35–42.

[40] Shaker H Ali El-Sappagh, Abdeltawab M Ahmed Hendawi, and Ali Hamed El Bastawissy. "A proposed model for data warehouse ETL processes." In: *Journal of King Saud University-Computer and Information Sciences* 23.2 (2011), pp. 91–104.

[41] *Social Media Profile Pictures – How Often Do You Change Yours?* https://www.wittysparks.com/social-media-profile-pictures-how-often-do-you-change-yours/. Accessed: 03.04.2018.

[42] Martin Staudt, Anca Vaduva, and Thomas Vetterli. "Metadata Management and Data Warehousing." In: 17 (1999).

[43] Ann Taylor, Mitchell Marcus, and Beatrice Santorini. "The Penn treebank: an overview." In: *Treebanks* (2003), pp. 5–22.

[44] *The quality of Twitter's users is deteriorating*. http://www.businessinsider.com/twitter-monthly-active-users-2015-7?r=UK&IR=T. Accessed: 30.03.2018.

[45] *TSIMMIS*. http://infolab.stanford.edu/tsimmis/. Accessed: 22.03.2018.

[46] *Twitter API Docs - Account and User*. https://developer.twitter.com/en/docs/accounts-and-users/follow-search-get-users/api-reference. Accessed: 07.03.2018.

[47] *Twitter - Rate Limits*. https://dev.twitter.com/web/sign-inhttps://dev.twitter.com/rest/public/rate-limits. Accessed: 07.03.2018.

[48] Panos Vassiliadis et al. "A Framework for the Design of ETL Scenarios." In: *International Conference on Advanced Information Systems Engineering* (2003), pp. 520–535.

[49] *Was ist Data Quality?* https://www.bigdata-insider.de/was-ist-data-quality-a-649900/. Accessed: 03.04.2018.

[50] *What percentage of people change their email address annually?* https://www.quora.com/What-percentage-of-people-change-their-email-address-annually. Accessed: 03.04.2018.

[51] *Why Use Python?* https://www.fullstackpython.com/why-use-python.html. Accessed: 02.04.2018.

[52] *Xing - API closes news*. https://www.xing.com/communities/posts/xing-news-die-oeffentlich-nutzbare-api-wird-geschlossen-1012410278. Accessed: 06.03.2018.

[53] *Xing - API Docs - Find users by keywords*. https://dev.xing.com/docs/get/users/find. Accessed: 06.03.2018.

[54] *Xing - Discontinue free public API*. http://www.nextscripts.com/updts/news/2017/01/xing-decided-discontinue-free-public-api. Accessed: 06.03.2018.