ΟΙΚΟΝΟΜΙΚΟ
ΠΑΝΕΠΙΣΤΗΜΙΟ
ΑΘΗΝΩΝ

ATHENS UNIVERSITY
OF ECONOMICS
AND BUSINESS

ATHENS UNIVERSITY
OF ECONOMICS AND BUSINESS
DEPARTMENT OF INFORMATICS
MSc IN INFORMATION SYSTEMS

THESIS

“Creating a repository for data crawled from multiple social networks”

KONSTANTINOS CHRISTOFILOS
MM4140023

ATHENS, OCT. 2016

AKNOWLEDGMENTS

This thesis was concluded as a part of AUEB's master in science program under the supervision of professor Ioannis Kotidis.

I want to thank Prof. Kotidis for giving me the opportunity to make a study in the, very fascinating, area of big data. Along with Prof. Kotidis, I would like to thank Mr. Vasilios Spiropoulos (Phd candidate) who was always there to assist me in any problem I faced.

I, also, want to thank all of my professors and colleagues, who made the last two years of my life a rollercoaster of fun and knowledge. I, really, owe them a lot.

Last, I want to thank the person who made all that possible, the person who pushed me to start, continue and finish that MSc. I want to thank my wife Efi and dedicate that thesis to her, because without her, that paper wouldn't have existed.

Table of Contents

ABSTRACT

People use countless web services in their everyday life. The same thing happens to companies and organizations. Most cloud services give access to the generated data via REST APIs, but although this seems very nice, it becomes a headache when someone wants to get results from more than one service.

The main problem is that API responses are not compatible among different services and the solution to that is the transformation of these semi-structured data to structured data.

The second problem, and probably the most hard to tackle, is to find relations in these data and produce some worthy results.

The approach that was adopted on this thesis was to fetch data from two input API sources (Twitter, Instagram), find relations between the data using natural language processing in the responses and finally merge them into a structured data environment. In that case, we manage to have a single query interface on the raw data and in addition, some relations pre-populated.

The merged output will be stored in two different graph databases, one property graph (Neo4J) and one semantic graph (RDF-Apache Jena). In that way we had the ability to give a unified query environment by using one of each databases.

Introduction

The rise of social networks, and web services in general, over the last decade have skyrocket the daily data production within the Internet. These vast amounts of daily-generated data created a great area of interest for everyone who is related in big data analytics.

All social networks and major web services give access to their data via REST APIs as JSON or XML responses. This is something very useful, because they expose their data to any kind of application needs them without having knowledge of each service's underlying technology.

All that sounds really great. Web services give access to their data, so anyone can analyze them and make decisions based on that analysis. So, where is the problem? The problem is that there isn't a dominant social network or web service, which means that most people or organizations use multiple networks/services to interact with others. For example, o politician can use all major social networks like Facebook, Twitter, Instagram along with Google Analytics to track its website traffic. All information that can be retrieved from these services doesn't follow the same schema, making the data analysis a real headache.

The first problem that we have to overcome is the difference in each API response format. We can handle that issue by transforming each API response from semi-structured to structured data, using a database to store them. That approach will, also, give us the ability to query the final result using the query language of the selected database.

The second, and probably the most hard to tackle, problem is the way we can relate these apparently unrelated data. That problem lies on the very own architecture of REST APIs, which is the abstraction of the underlying technology. Each service has a different entity model and forms relationships between their entities in its own way. Besides that, it's not responsible to know the schema of other services. That means, that the same user in Twitter cannot be identified in Instagram or Facebook.

The solution that was applied for that problem was to make relation paths by reference. We tried to find entities that each API response refers and connect different API responses via these references.

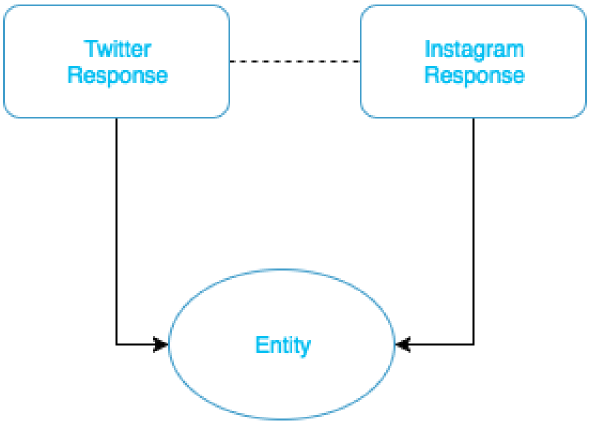


Figure 1 - Relation build between different APIs

In more detail, we mapped some responses from the APIs that we will put under test and tried to find entities on each response using natural language processing algorithm for entity recognition. The recognized entities are related to each response, which in turn create relation paths between responses of separate APIs. The final result is stored in a database.

Since, we wanted to create relations and paths between entities, we selected for storage two graph databases. An RDF triplestore (Apache Jena) and a property graph database (Neo4J).

We have the ability to store the generated data in each or both of them at the same time, giving the ability to experiment in two very well known graph models.

Another reason that two different graph models were selected was to give the option for anyone who wanted to analyze the generated data to use the query language that is more fluent in.

The algorithm that is used for entity recognition in API responses is the Stanford NER [1], which is a Java implementation of a named entity recognizer that parses plain text and identifies entities such as persons, locations and organizations. The library can be extended for more entities but we used the default English implementation since it is out of that thesis scope the field of named entity recognition.

The last parameter that was taken in account in the software design was that the application should be as easy as possible in use. That's why the process of data aggregation is separated in two steps.

First we feed the application with a list of names in a plain text file and it populates a list of API endpoints for each service that we want to fetch data, based on that name list.

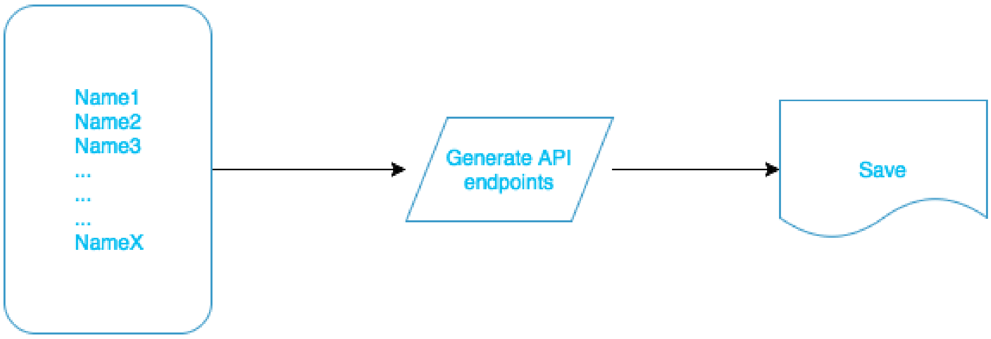


Figure 2 - Endpoint generation by name list

The second step is to run in specified time intervals, based on the expected volume, a batch process that will fetch, relates and stores the final data.

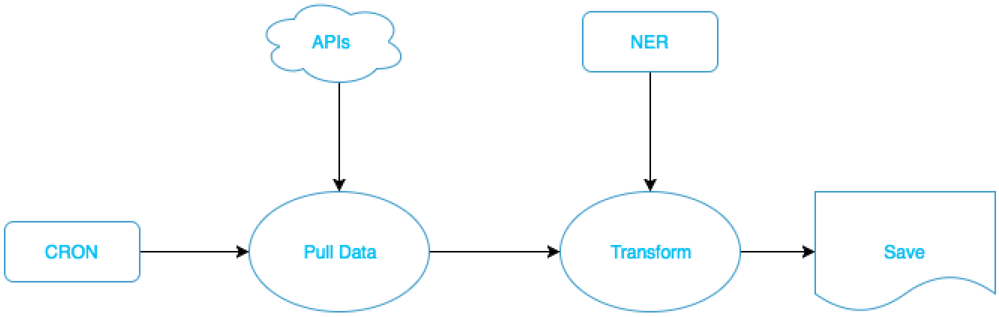


Figure 3 - Flow of API data migration

Chapter 1: REST APIs

Representational State Transfer (REST)

REST stands for Representational State Transfer and is an abstraction of the architectural elements within a distributed hypermedia system.

REST ignores the details of component implementation and protocol syntax in order to focus on the roles of components, the constraints upon their interaction with other components, and their interpretation of significant data elements. [2] [3]

In short terms REST is an architecture that enables applications to communicate without knowing the technology each one uses and even exist in the same physical machine. Its purpose is to induce performance, scalability, simplicity, modifiability, visibility, portability and reliability [2] [3]

Application Programming Interface (API)

API is an Application Programming Interface, which is a set of routine definitions, protocols and tools for building software applications. In terms of programming languages, an API is the set of functions and commands that the language gives to developer to work with.

So, a REST API is, kind of, the communication language of servers running the World Wide Web. It can be accessed using Uniform Resource Identifiers (URIs) using the HTTP protocol and return resources in various formats (HTML, XML, JSON etc.)

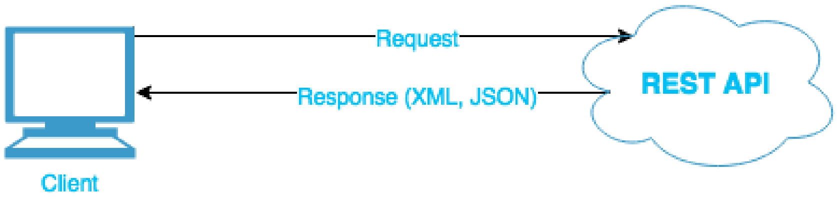


Figure 4 - REST API

Chapter 2: Graph databases

Graph database is a database type that uses graph structures with nodes, edges and properties to represent and store data. A key concept of the system is the graph, which directly relates data items in the store.

The underlying storage mechanism of different graph databases varies, from relational engines storing graph data in a table to a key-value store or document-based database. [4]

Graph databases are based on graph theory and employs nodes, edges and properties.

Nodes represent entities such as people, businesses, accounts, or any other item you might want to keep track of. They are roughly the equivalent of the record, relation or row in a relational database, or the document in a document database.

Edges, also known as graphs or relationships, are the lines that connect nodes to other nodes and represent relationship between them.

Properties are pertinent information that relate to nodes. For instance, if Wikipedia were one of the nodes, one might have it tied to properties such as “website”, “reference material” etc. [5]

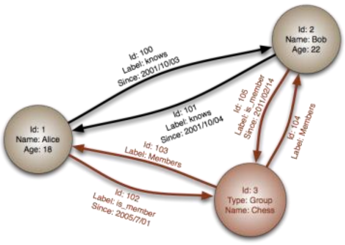


Figure 5 - Graph

Resource Description Framework (RDF)

RDF is a standard model for data interchange on the Web and was specified by W3C. Since RDF was created to “describe” Web, it made a perfect sense to be used as a graph data structure to store data. Web is a graph, created by nodes, edges and relations. [6]

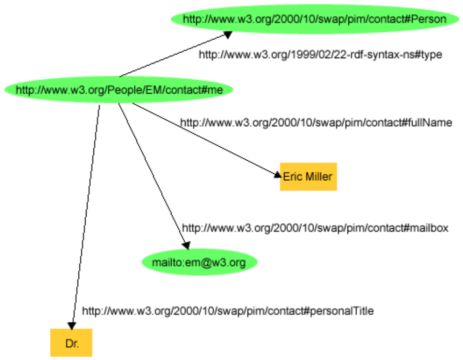

http://www.w3.org/2000/10/swap/pim/contact#Person
http://www.w3.org/1999/02/22-rdf-syntax-ns#type
http://www.w3.org/People/EM/contact#me
http://www.w3.org/2000/10/swap/pim/contact#fullName
Eric Miller
http://www.w3.org/2000/10/swap/pim/contact#mailbox
mailto:em@w3.org
http://www.w3.org/2000/10/swap/pim/contact#personalTitle
Dr.

Figure 6 - RDF Scheme [7]

Property Graph

Property Graph databases are graph databases that contains connected entities, which can cold, any number of attributes (key-value pairs). Nodes can be tagged with labels representing their different roles in the defined domain. In addition to contextualizing node and relationship properties, labels may also serve to attach metadata to certain nodes. [8]

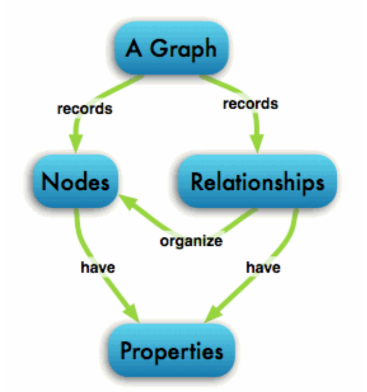


Figure 7 - Building blocks of the Property Graph [8]

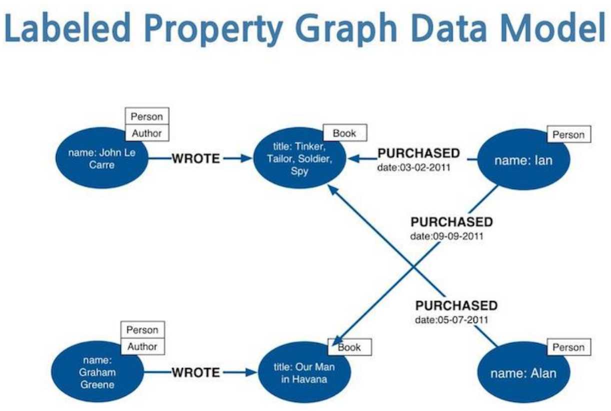


Figure 8 - Property Graph Model [8]

Chapter 3: Natural Language Processing (NLP)

Natural language processing (NLP) is an area of research and application that explores how computers can be used to understand and manipulate natural language text or speech to do useful things.

NLP lie in a number of disciplines, information sciences, linguistics, mathematics, electrical and electronic engineering, artificial intelligence and robotics, psychology, etc. Applications of NLP include a number of fields of studies, such as machine translation, natural language text processing and summarization, user interfaces, multilingual and cross language information retrieval, speech recognition, artificial intelligence and expert systems, and so on. [9]

Named Entity Recognition (NER)

Named entity recognition (NER) is a subtask of information extraction that seeks to locate and classify named entities in text into pre-defined categories such as the names of persons, organizations, locations, expressions of time etc. NER systems use linguistic grammar-based techniques as well as statistical models, i.e. machine learning. [10] [11] [12] [13]

Chapter 4: Architecture

The implementation of the thesis was designed based on the elements abstraction and scaling. The architecture that was adopted was service oriented (Service Oriented Architecture – SOA) exposing services for each database and the application components. That gives us the ability to scale our application and use more machines that can work in parallel.

The software that was created has a backend part that handles all the processing and a frontend user interface that is used to configure the backend and run some limited actions.

Programming Language

The programming language that was used for the implementation was PHP 7. The reason this language was selected is that it can provide all the needed functionality and, also, there were a lot of ready to use libraries and bigger community.

The application was built using the Silex PHP micro-framework [13].

Environment

In order to test the abstraction of all components, the best scenario would be to have different physical machines for each part of the process. Because these resources weren't easy to be allocated in a development environment, we emulated that procedure with the use of Docker containers [14] and we created separated virtual machines for each needed component.

Named Entity Recognizer (NER)

The text process part of the software was made with the use of a named entity recognizer (NER). We use a Stanford University implementation in Java, which was created by "The Stanford Natural Language Processing Group" [1]

The NER parses the content of each REST API response and finds entities inside free text and connects these entities with the main object, which is the response itself.

Chapter 5: APIs (Inputs) / Databases (Outputs)

Inputs

Each API is defined as an input for the application. We created two different type of inputs based on the authentication protocol each API is using. Specifically we have OAuth1 [15] and OAuth2 [16] input types.

The inputs we are using for the experiments (Facebook, Instagram and Twitter) are inherited from these base input types.

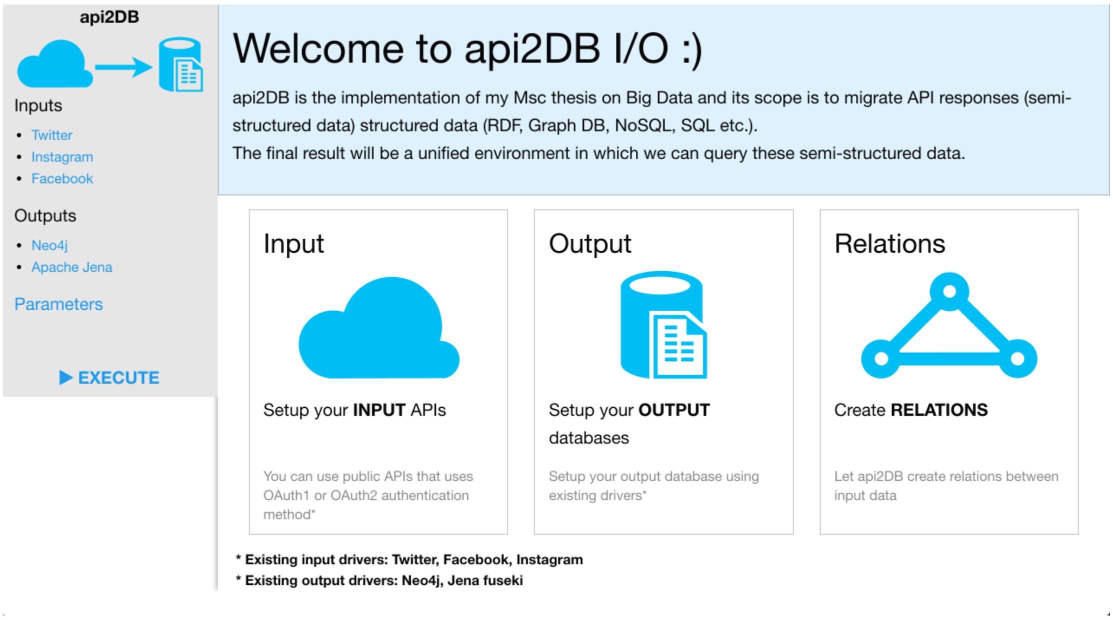
Outputs

On the other side, each database used to store data is defined as output. The outputs are designed so they can send data to rest services, instead of using direct connections to each database. This is very useful because it inserts a level of abstraction in the data save process, giving the option to create more outputs to send data in other types of storage.

Chapter 6: Implementation

User interface

The application has a Web interface that serves two purposes. The first is to configure the application and the other is to run custom calls, to test how to integrate with the databases.



Welcome to api2DB I/O :)

api2DB is the implementation of my Msc thesis on Big Data and its scope is to migrate API responses (semi-structured data) structured data (RDF, Graph DB, NoSQL, SQL etc.).
The final result will be a unified environment in which we can query these semi-structured data.

Setup your **INPUT** APIs

You can use public APIs that uses OAuth1 or OAuth2 authentication method*

Setup your **OUTPUT** databases

Setup your output database using existing drivers*

Create **RELATIONS**

Let api2DB create relations between input data

*** Existing input drivers: Twitter, Facebook, Instagram**
*** Existing output drivers: Neo4j, Jena fuseki**

Figure 9 - Application user interface (Homepage)

Homepage
From the left sidebar, we can configure our inputs, outputs, generic configuration for the application and execute the manual inputs we have defined in the web interface.

Configuration

All configuration that can be processed through the Web interface, is saved in a YAML [17] file like this:

```
inputs:
    twitter:
        name: Twitter
        oauth_type: '1'
        credentials:
            identifier: DVXHssj5CskPZaSlTPBpqpkzY
            secret:
CaEIf10GgyGQc1U7IsCVWtQA3t152PAHD1S99zMqzleUyIYvkw
            client_key: 265929407-
dkT7Hm8R7NAdDHFKcKbW7ZgBjTVWYA621pg2UW7H
            client_secret:
zJyXL7il0u8MxABBCc3qMaupXIwKsx72oUj0a1VhDemLc
    instagram:
        name: Instagram
        oauth_type: '2'
        credentials:
            access_token:
1798817488.fc65250.50fc018193734f38b8b92a19d9c6bfd4
    facebook:
        name: Facebook
        oauth_type: '2'
        credentials:
            access_token:
EAALsQTxxj0YBADBfE6pjZAItsRZAb6OQqR6cDP9NjYDpTT1ROCUaMZBtkCROwZAUXkOE
R2SXZC8WZAt6QZCDfOZC6fAZBKd5m58G8rYaaWFSVp1BM9DZABCZAbMMZCHaBfOcVoCXw
8HQMsL4jmuXmPykWyw0OvGkLe6EHfAZD
outputs:
    neo4j:
        name: Neo4j
        host: localhost
        port: 7474
        credentials:
            username: neo4j
            password: '123456'
    jena:
        name: 'Apache Jena'
        host: localhost
        port: 3030
        path: ds
parameters:
    ner_path: /Users/kostasx/Downloads/stanford-ner-2015-12-09
```

Inputs

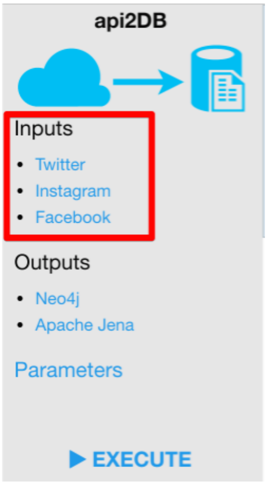
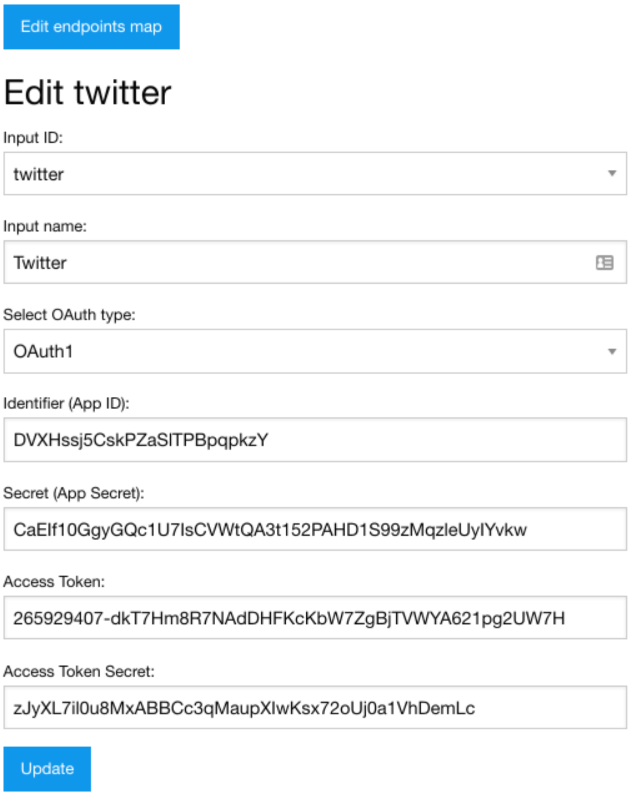


Figure 10 - Application user interface (Inputs)

Each input has different configuration and it's different based on the authentication method they use.



Edit twitter

Input ID: twitter

Input name: Twitter

Select OAuth type: OAuth1

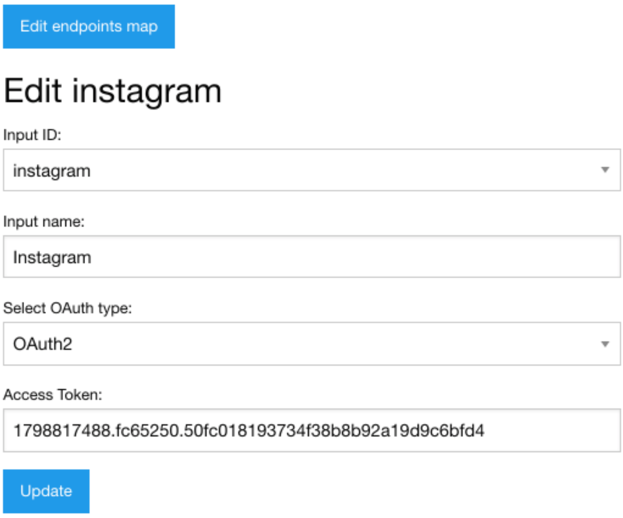
Identifier (App ID): DVXHssj5CskPZaSlTPBpqpkzY

Secret (App Secret): CaElf10GgyGQc1U7lsCVWtQA3t152PAHD1S99zMqzleUylYvkw

Access Token: 265929407-dkT7Hm8R7NAdDHFKcKbW7ZgBjTVWYA621pg2UW7H

Access Token Secret: zJyXL7il0u8MxABBCc3qMaupXlwKsx72oUj0a1VhDemLc

Figure 11 - Application user interface (OAuth1 Input)



Edit instagram

Input ID: instagram

Input name: Instagram

Select OAuth type: OAuth2

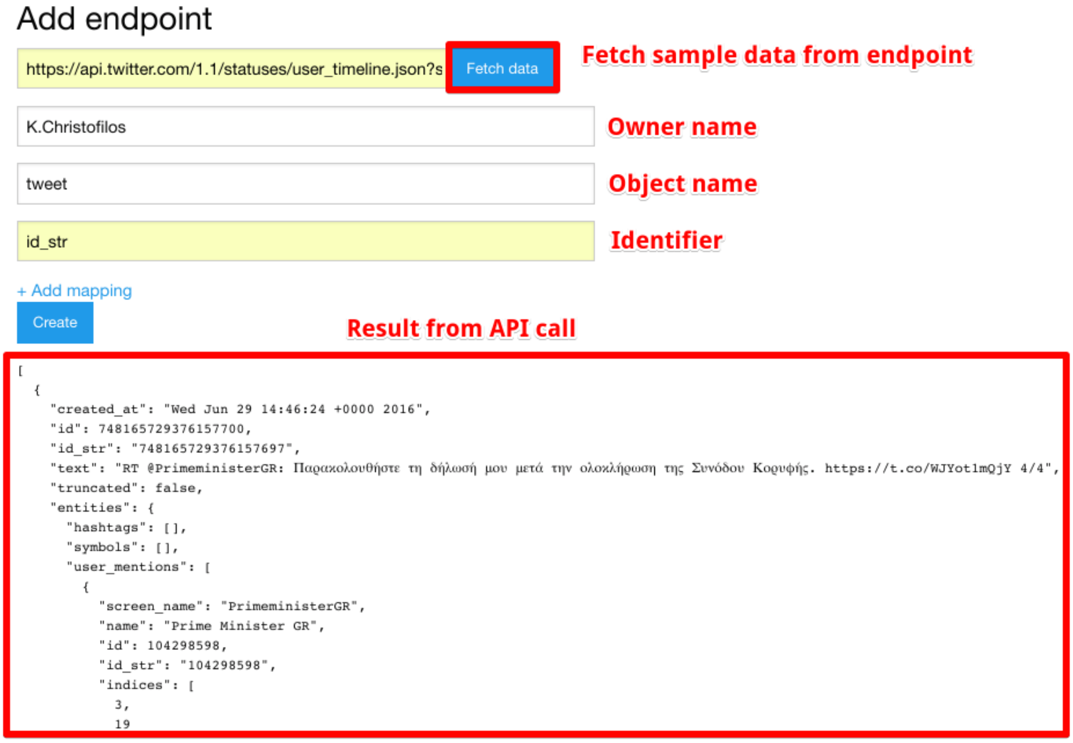
Access Token: 1798817488.fc65250.50fc018193734f38b8b92a19d9c6bfd4

Figure 12 - Application user interface (OAuth2 Input)

By clicking the “Edit endpoints map” the user can add and map API endpoints to the list of manual endpoints that will be run when the “Execute” in the left column is pressed.

In order to enter a new endpoint you have to define the endpoint, the name of the endpoint Owner, the name of the object that we will consider that endpoint to be and a unique identifier from the response data.

All endpoints, along with their parameters are saved in a YAML[15] file.



https://api.twitter.com/1.1/statuses/user_timeline.json?s

Fetch sample data from endpoint

K.Christofilos

Owner name

tweet

Object name

id_str

Identifier

Result from API call

```
[
  {
    "created_at": "Wed Jun 29 14:46:24 +0000 2016",
    "id": 748165729376157700,
    "id_str": "748165729376157697",
    "text": "RT @PrimeministerGR: Παρακολουθήστε τη δήλωσή μου μετά την ολοκλήρωση της Συνόδου Κορυφής. https://t.co/WJYot1mQjY 4/4",
    "truncated": false,
    "entities": {
      "hashtags": [],
      "symbols": [],
      "user_mentions": [
        {
          "screen_name": "PrimeministerGR",
          "name": "Prime Minister GR",
          "id": 104298598,
          "id_str": "104298598",
          "indices": [
            3,
            19
```

Figure 13 - Application user interface (Add endpoint)

Outputs

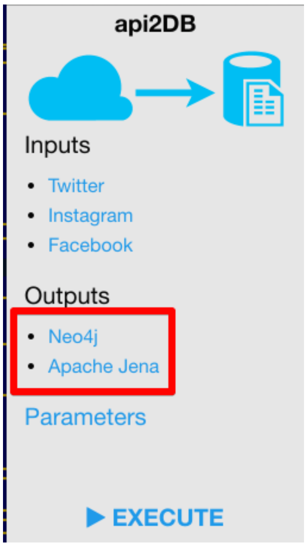
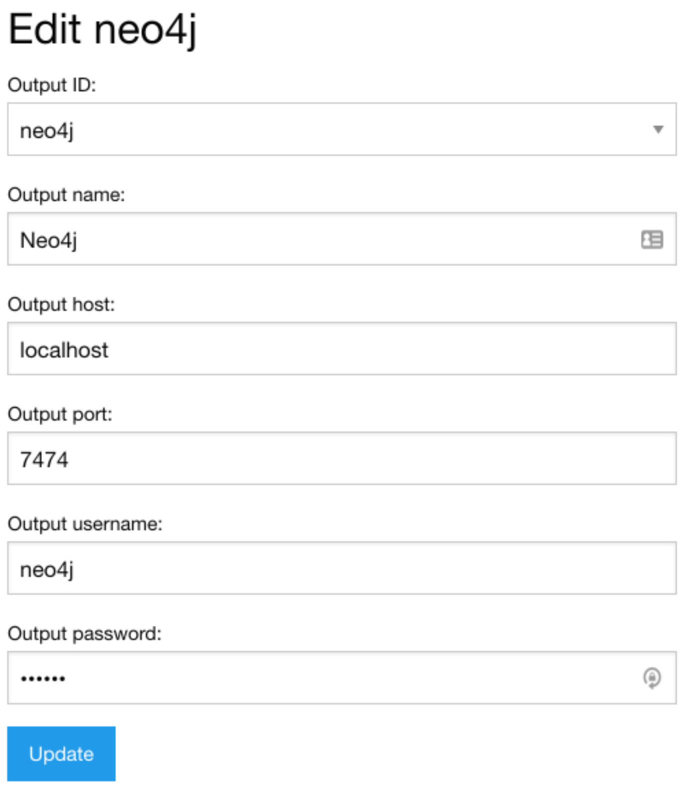


Figure 14 - Application user interface (Outputs)

Each output has the same configuration with small alterations based on database needs.



Edit neo4j

Output ID: neo4j

Output name: Neo4j

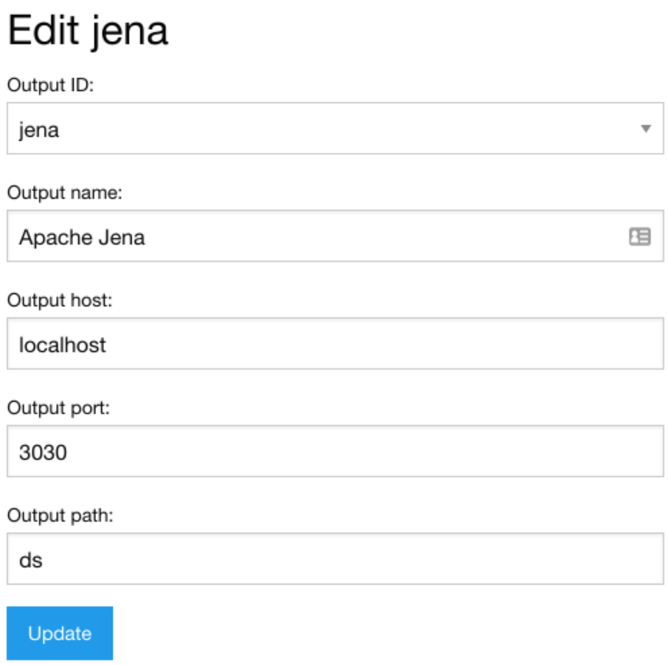
Output host: localhost

Output port: 7474

Output username: neo4j

Output password: ••••••

Figure 15 - Application user interface (Neo4j)



Edit jena

Output ID: jena

Output name: Apache Jena

Output host: localhost

Output port: 3030

Output path: ds

Figure 16 - Application user interface (Apache Jena)

Parameters

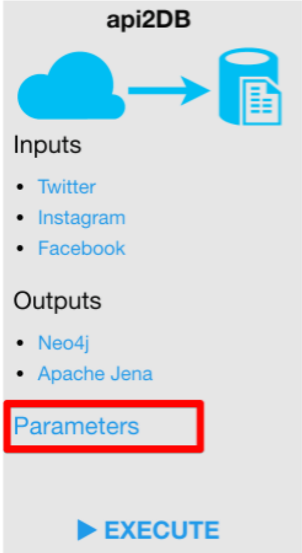
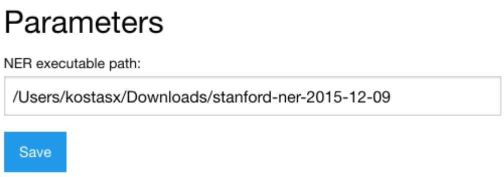


Figure 17 - Application user interface (Parameters)

This option has generic configuration for the application:

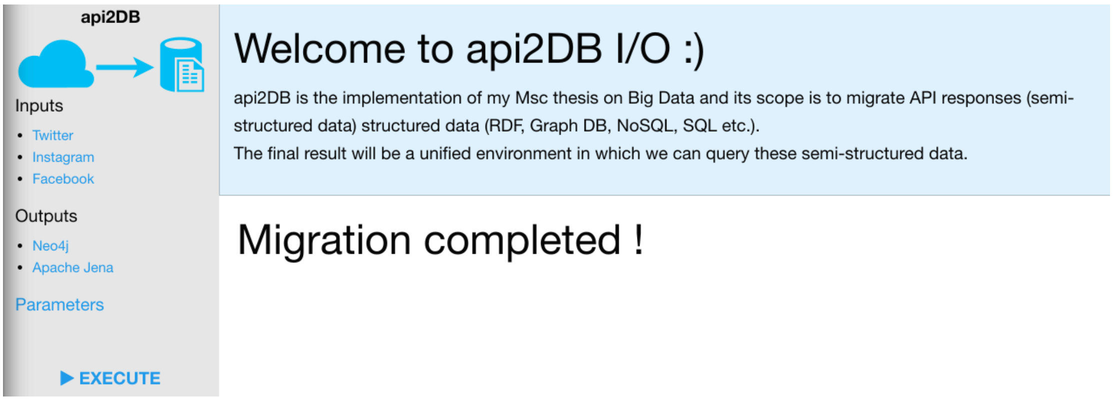

Parameters

NER executable path: /Users/kostasx/Downloads/stanford-ner-2015-12-09

Figure 18 - Application user interface (Generic configuration)

Execute

By pressing the “EXECUTE” link the application will start fetching data from the endpoints that are manually defined and will migrate the data to the outputs that are configured.



Welcome to api2DB I/O :)

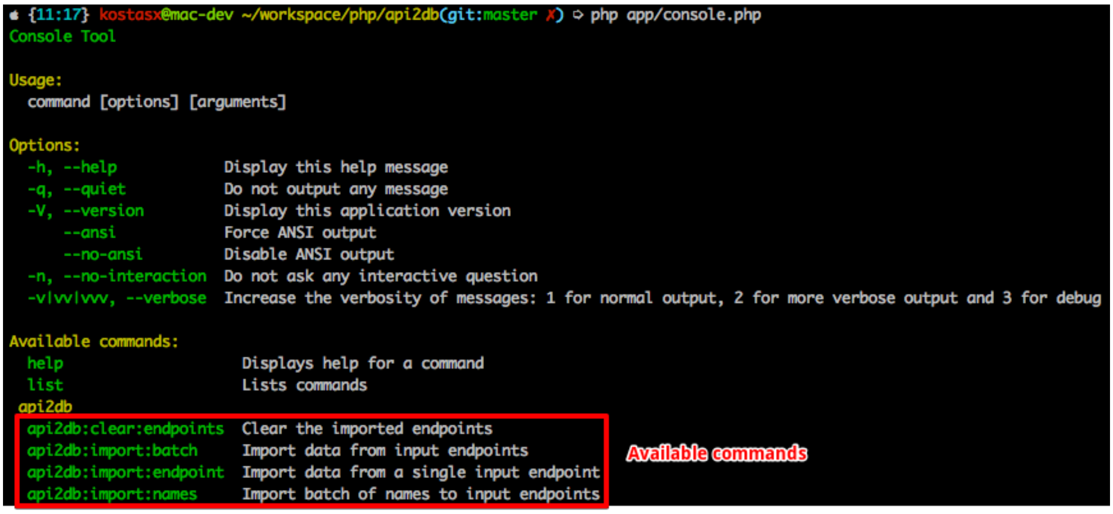
api2DB is the implementation of my Msc thesis on Big Data and its scope is to migrate API responses (semi-structured data) structured data (RDF, Graph DB, NoSQL, SQL etc.).
The final result will be a unified environment in which we can query these semi-structured data.

Migration completed !

Figure 19 - Application user interface (Execute)

Command line interface (CLI)

Most of the application's functionality is build in Command Line Interfaces (CLI), which are accessed through a single file, the `*app/console.php*`. That file registers all separate CLI applications and can run them from a single interface.

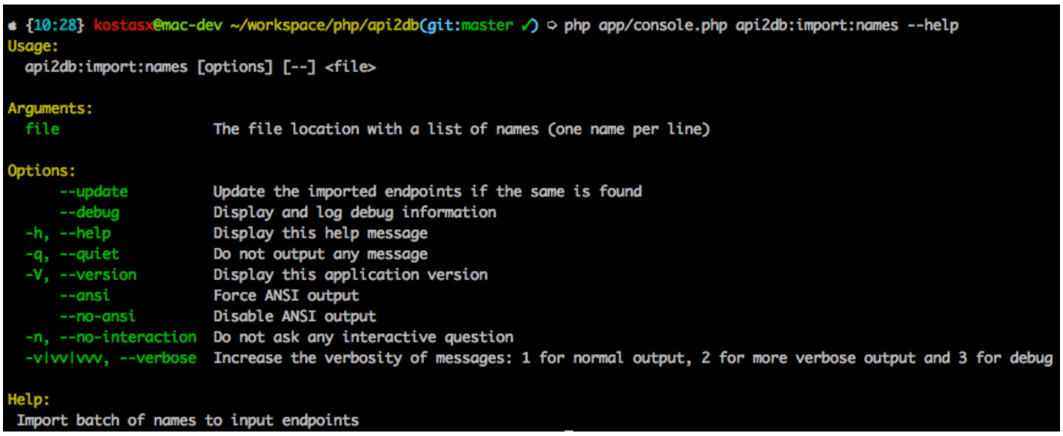


```
 {11:17} kostasx@mac-dev ~/workspace/php/api2db(git:master ✗) ➭ php app/console.php
Console Tool

Usage:
  command [options] [arguments]

Options:
  -h, --help            Display this help message
  -q, --quiet           Do not output any message
  -V, --version         Display this application version
      --ansi            Force ANSI output
      --no-ansi         Disable ANSI output
  -n, --no-interaction  Do not ask any interactive question
  -v|vv|vvv, --verbose  Increase the verbosity of messages: 1 for normal output, 2 for more verbose output and 3 for debug

Available commands:
  help                    Displays help for a command
  list                    Lists commands
 api2db
  api2db:clear:endpoints  Clear the imported endpoints
  api2db:import:batch     Import data from input endpoints
  api2db:import:endpoint  Import data from a single input endpoint
  api2db:import:names     Import batch of names to input endpoints
```

Figure 20 - Application CLI (Available commands)

With the available CLI commands we can transform a list of names into Input API endpoints, we can clear that list and we can migrate the data that will come from the inputs to the outputs after we relate them using the NER.

Generate endpoints from names list (api2db:import:names)



```
 {10:28} kostasx@mac-dev ~/workspace/php/api2db(git:master ✓) ➪ php app/console.php api2db:import:names --help
Usage:
  api2db:import:names [options] [--] <file>

Arguments:
  file                      The file location with a list of names (one name per line)

Options:
      --update              Update the imported endpoints if the same is found
      --debug               Display and log debug information
  -h, --help                Display this help message
  -q, --quiet               Do not output any message
  -V, --version             Display this application version
      --ansi                Force ANSI output
      --no-ansi             Disable ANSI output
  -n, --no-interaction      Do not ask any interactive question
  -v|vv|vvv, --verbose      Increase the verbosity of messages: 1 for normal output, 2 for more verbose output and 3 for debug

Help:
 Import batch of names to input endpoints
```

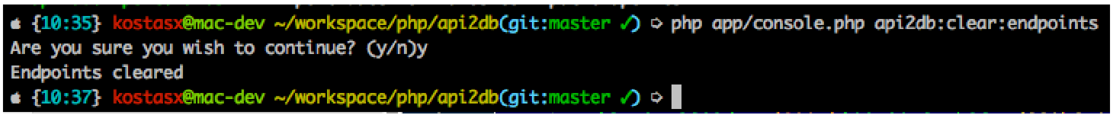
Figure 21 - Application CLI (Generate endpoints from names)

We can add in a text file a list of names and parse it through the application. The application will discover relevant accounts to the defined inputs and will generate a list of endpoints along with their owners and object types.

This makes it very easy to generate a list of probable endpoints based on some related names, giving the user the ability to, somewhat, query the inputs for relevant accounts.

The endpoints that will be generated by that command are separate from the endpoints that are defined in the web interface and are not mixed up.

Clear generated endpoints (api2db:clear:endpoints)



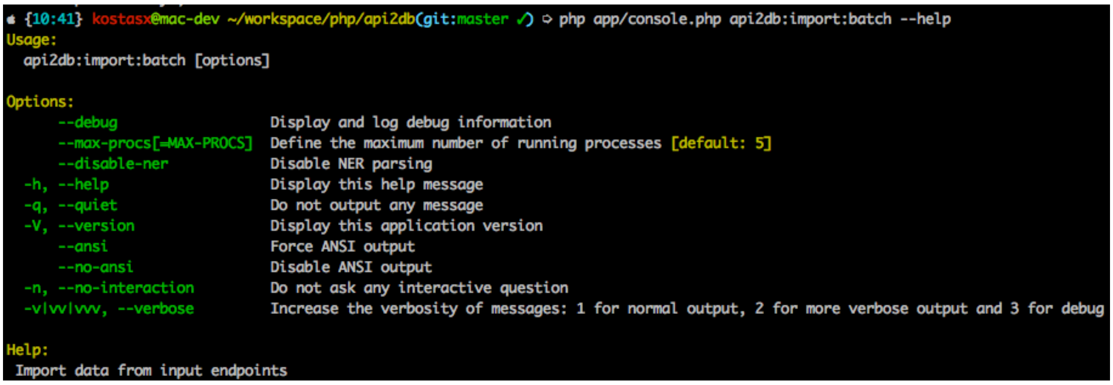
```
 {10:35} kostasx@mac-dev ~/workspace/php/api2db(git:master ✓) ➪ php app/console.php api2db:clear:endpoints
Are you sure you wish to continue? (y/n)y
Endpoints cleared
 {10:37} kostasx@mac-dev ~/workspace/php/api2db(git:master ✓) ➪ █
```

Figure 22 - Application CLI (Clear endpoints)

That command clears all the generated endpoints from the map file.

Batch import data from endpoints (api2db:import:batch)

That command is the actual core of the application. It fetches data from the, previously, imported endpoints, passes the data through the NER and migrates the data to the outputs.



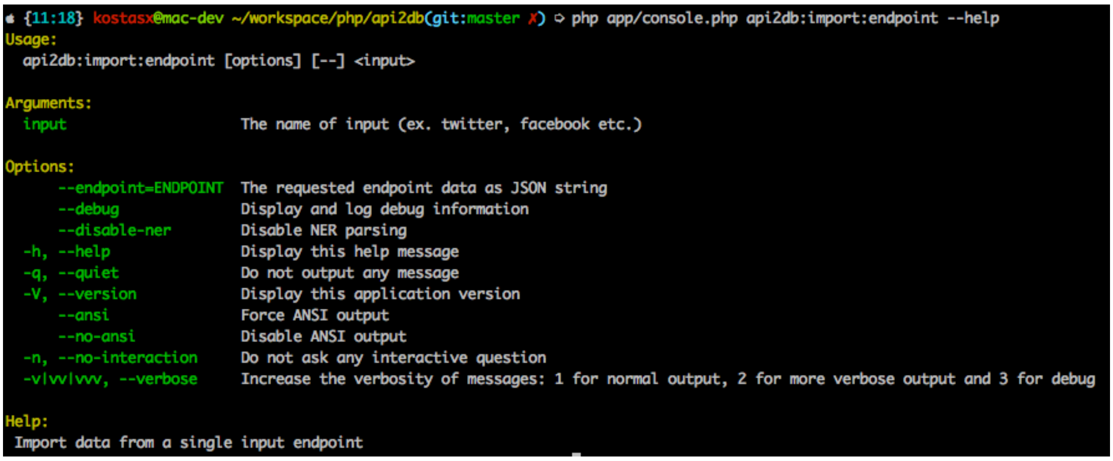
```
 {10:41} kostasx@mac-dev ~/workspace/php/api2db(git:master ✓) ➪ php app/console.php api2db:import:batch --help
Usage:
  api2db:import:batch [options]

Options:
      --debug                  Display and log debug information
      --max-procs[=MAX-PROCS]  Define the maximum number of running processes [default: 5]
      --disable-ner            Disable NER parsing
  -h, --help                   Display this help message
  -q, --quiet                  Do not output any message
  -V, --version                Display this application version
      --ansi                   Force ANSI output
      --no-ansi                Disable ANSI output
  -n, --no-interaction         Do not ask any interactive question
  -v|vv|vvv, --verbose         Increase the verbosity of messages: 1 for normal output, 2 for more verbose output and 3 for debug

Help:
 Import data from input endpoints
```

Figure 23 - Application CLI (Batch import from endpoints)

While running that command you can define the maximum number of concurrent processes that will be run with the parameter `*--max-procs[=MAX-PROCS]*`. That option was required because the application can run in different type of machines with different abilities.

Also, there is the option `*--disable-ner*` that disables the NER parsing from the input data. This can be used, when a user wants to test the inputs themselves and remove NER from the process.

Import data from a single endpoint (api2db:import:endpoint)



```
 {11:18} kostasx@mac-dev ~/workspace/php/api2db(git:master ✗) ➪ php app/console.php api2db:import:endpoint --help
Usage:
  api2db:import:endpoint [options] [--] <input>

Arguments:
  input                    The name of input (ex. twitter, facebook etc.)

Options:
      --endpoint=ENDPOINT  The requested endpoint data as JSON string
      --debug              Display and log debug information
      --disable-ner        Disable NER parsing
  -h, --help               Display this help message
  -q, --quiet              Do not output any message
  -V, --version            Display this application version
      --ansi               Force ANSI output
      --no-ansi            Disable ANSI output
  -n, --no-interaction     Do not ask any interactive question
  -v|vv|vvv, --verbose     Increase the verbosity of messages: 1 for normal output, 2 for more verbose output and 3 for debug

Help:
 Import data from a single input endpoint
```

Figure 24 - Application CLI (Single endpoint import)

That command is a actually a sub-process of the api2db:import:batch, but it can be executed on its own.

It accepts the same options as the batch command, except the maximum processes option, witch doesn't apply here.

Chapter 7: Results

The experiment was made using the names of 2016 world's greatest leaders, based on Fortune magazine's list [18].

The list of the 56 names is the following:

Jeff Bezos, Angela Merkel, Aung San Suu Kyi, Pope Francis, Tim Cook, John Legend, Christina Figueres, Paul Ryan, Ruth Bader Ginsburg, Sheikh Hasina, Nick Saban, Huateng "Pony" Ma, Sergio Moro, Bono, Stephen Curry, Steve Kerr, Bryan Stevenson, Nikki Haley, Lin-Manuel Miranda, Marvin Ellison, Reshma Saujani, Larry Fink, Scott Kelly, Mikhail Kornienko, David Miliband, Anna Maria Chavez, Carla Hayden, Maurizio Macri, Alicia Garza, Patrisse Cullors, Opal Tometi, Chai Jing, Moncef Slaoui, John Oliver, Marc Edwards, Arthur Brooks, Rosie Batty, Kristen Griest, Shaye Haver, Denis Mukwege, Christine Lagarde, Marc Benioff, Gina Raimondo, Amina Mohammed, Domenico Lucano, Melinda Gates, Susan Desmond-Hellman, Arvind Kejriwal, Jorge Ramos, Michael Froman, Mina Guli, Ramon Mendez, Bright Simons, Justin Trudeau, Clare Rewcastle Brown, Tshering Tobgay.

The list of the previous names after it was parsed from the application produced a list of more than 11.000 endpoints in Facebook, Instagram and Twitter with the following distribution:

Names	Facebook Page Endpoints	Twitter Endpoints	Instagram Endpoints	Total Endpoints
56	437	866	9903	11206

Table 1 - API endpoint distribution

The first thing to be noticed is that there are much more Instagram accounts related to important people than on other social media. The reason for that is that the Instagram endpoints we used are these that have comments related to those people.

The parse of those endpoints in a single workstation[1] took about 14 days (period 2016-06-04 / 2016-06-18), with the most time consumed in the entity recognition process.

Performance

The application run a single pass over the generated endpoints witch took about 14 days in a single workstation and the generated nodes were 126,395.

Endpoints	Machines	Generated nodes	Time	Average nodes/day/machine
11206	1	126395	14 days	9028

Table 2 - Application performance

The time to insert a dataset from an endpoint to a database is displayed below in two stages. The first is the average time to insert data to each database when databases where empty and the second when the databases had more than 100,000 nodes

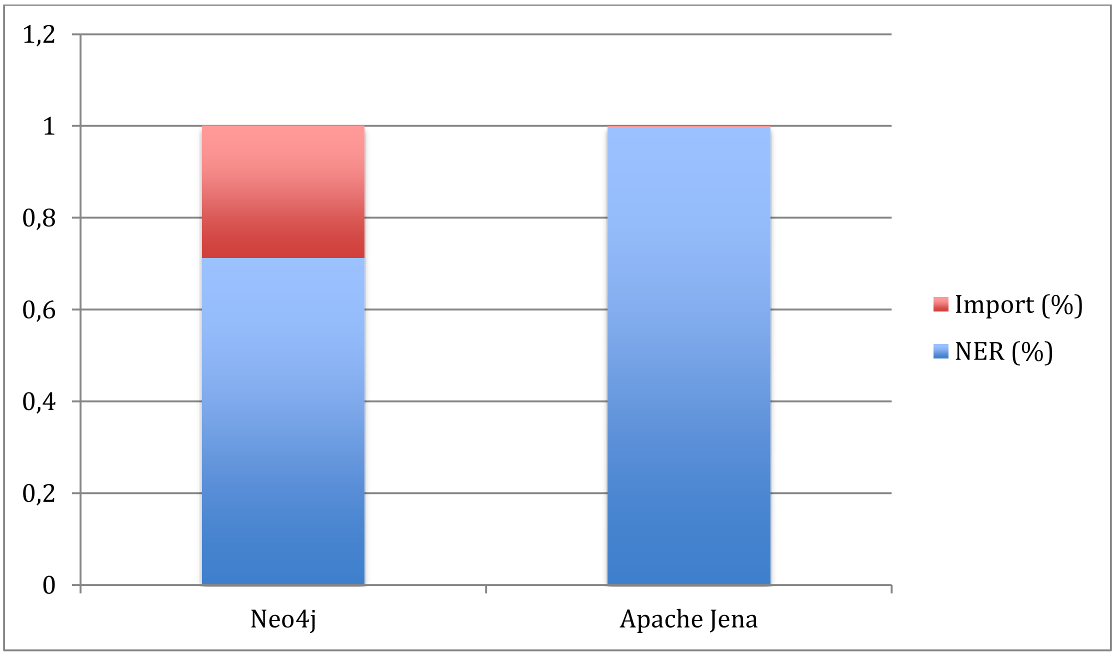


Diagram 1 - Data import time distribution (Empty database)

---

[1] Workstation specs: AMD FX 2-Core CPU, 4GB RAM, 120GB SSD, Linux OS with 5 concurrent processes of the application running

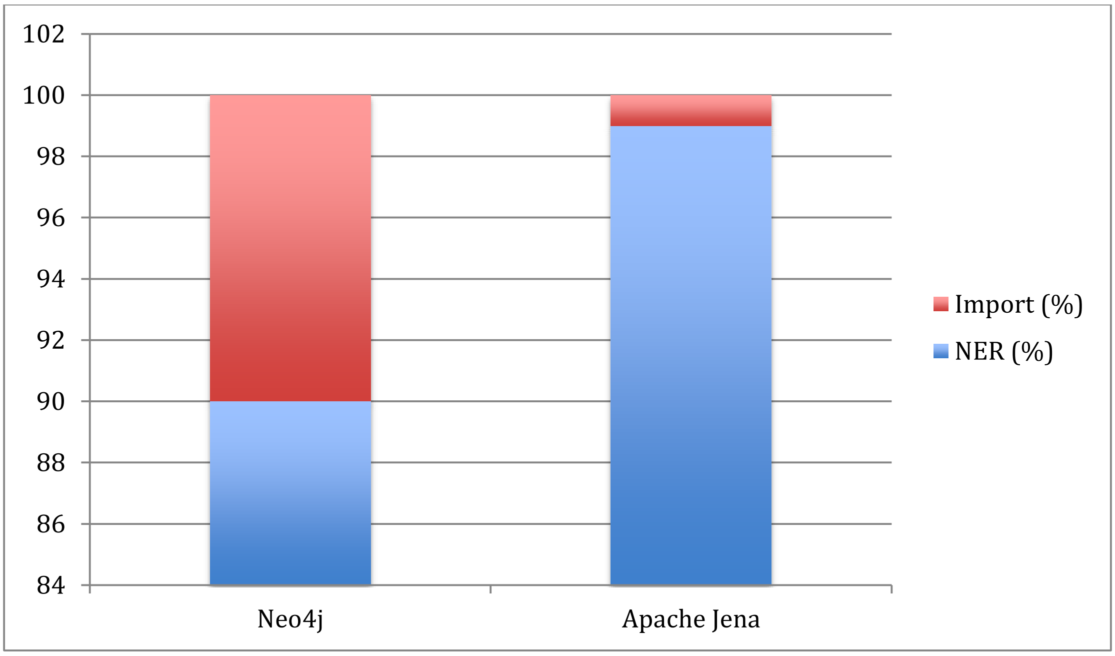


Diagram 2 - Data import time distribution (100,000+ nodes)

By watching these diagrams, it's more than clear that the most time of the process is consumed in the entity recognizer. About the two different databases, it seems that Neo4j takes much more time to insert the same data than the RDF store.

The reason for that is that in order to avoid duplicate entries in Neo4j, we have to check first if the nodes and the relations exist already and then insert the new data. On the other side, Jena doesn't require such a control, since the query to insert data is the same for insert and/or update.

Generated data

Below we show the total of discovered entities from the NER algorithm, along with their total relations with API responses. These diagrams give us a sample of what the application can do in a matter of pure entity numbers.

Below each diagram there is the according queries for Neo4J and Apache Jena

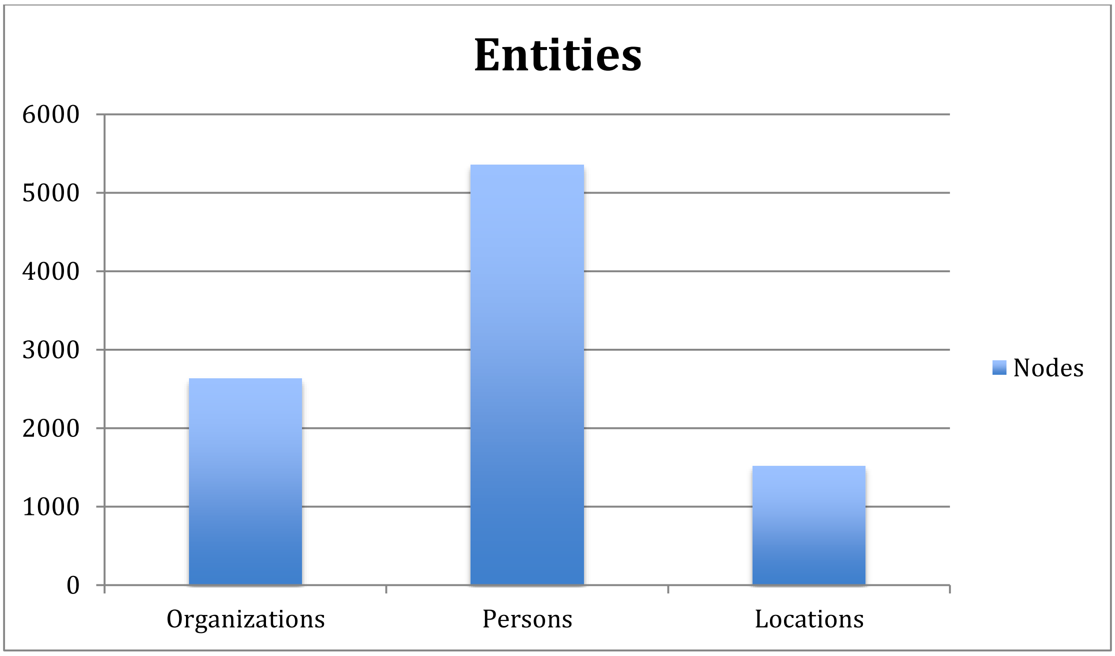


Diagram 3 - Entities that was recognized from API data

Neo4J Queries

```
MATCH (n:ORGANIZATION) RETURN count(n) as total_organizations

MATCH (n:PERSON) RETURN count(n) as total_persons

MATCH (n:LOCATION) RETURN count(n) as total_locations
```

Apache Jena: Query

```
PREFIX relation_ns: <http://custom/ns/relations#>

SELECT (COUNT(DISTINCT ?entity) AS ?total_organizations)
WHERE {
        ?subject relation_ns:REFERS_TO_ORGANIZATION ?entity
}

PREFIX relation_ns: <http://custom/ns/relations#>

SELECT (COUNT(DISTINCT ?entity) AS ?total_persons)
WHERE {
        ?subject relation_ns:REFERS_TO_PERSON ?entity
}

PREFIX relation_ns: <http://custom/ns/relations#>

SELECT (COUNT(DISTINCT ?entity) AS ?total_locations)
WHERE {
        ?subject relation_ns:REFERS_TO_LOCATION ?entity
}
```

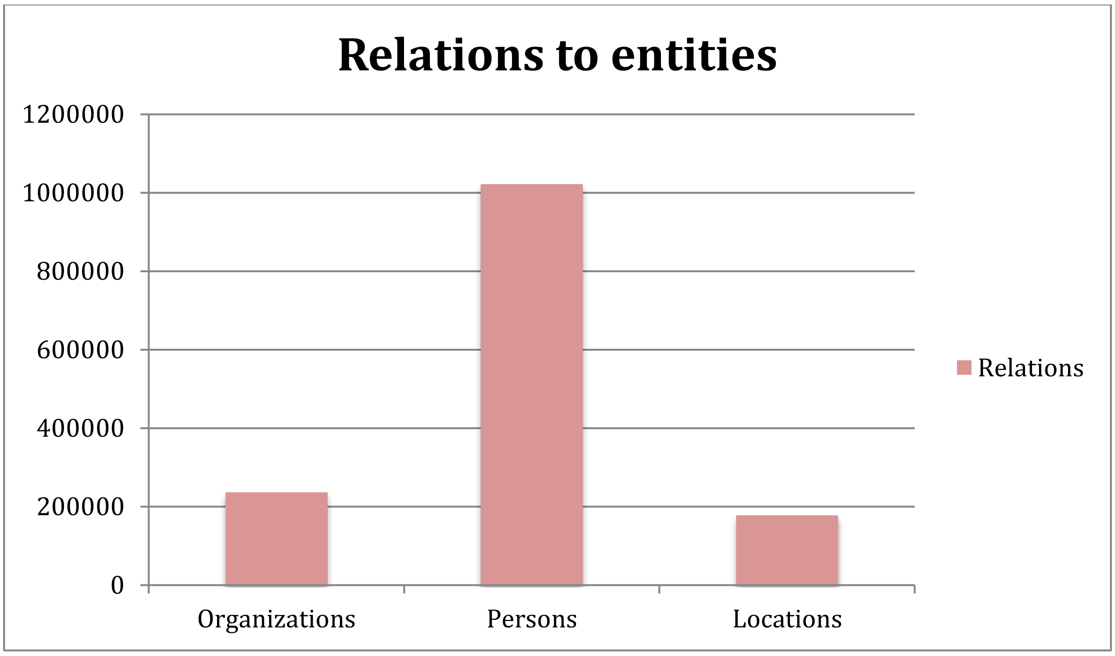


Diagram 4 - Relations to discovered entities

Neo4J Queries

```
MATCH (o)-[r]->(n:ORGANIZATION) RETURN count(r) as
total_relations_organization

MATCH (o)-[r]->(n:PERSON) RETURN count(r) as total_relations_person

MATCH (o)-[r]->(n:LOCATION) RETURN count(r) as total_relations_location
```

Apache Jena: Query

```
PREFIX relation_ns: <http://custom/ns/relations#>

SELECT (COUNT(?entity) AS ?total_relations_organization)
WHERE {
        ?subject relation_ns:REFERS_TO_ORGANIZATION ?entity
}

PREFIX relation_ns: <http://custom/ns/relations#>

SELECT (COUNT(?entity) AS ?total_relations_person)
WHERE {
        ?subject relation_ns:REFERS_TO_PERSON ?entity
}

PREFIX relation_ns: <http://custom/ns/relations#>

SELECT (COUNT(?entity) AS ?total_relations_location)
WHERE {
        ?subject relation_ns:REFERS_TO_LOCATION ?entity
}
```

Below we see the top 5 discovered entities by category:

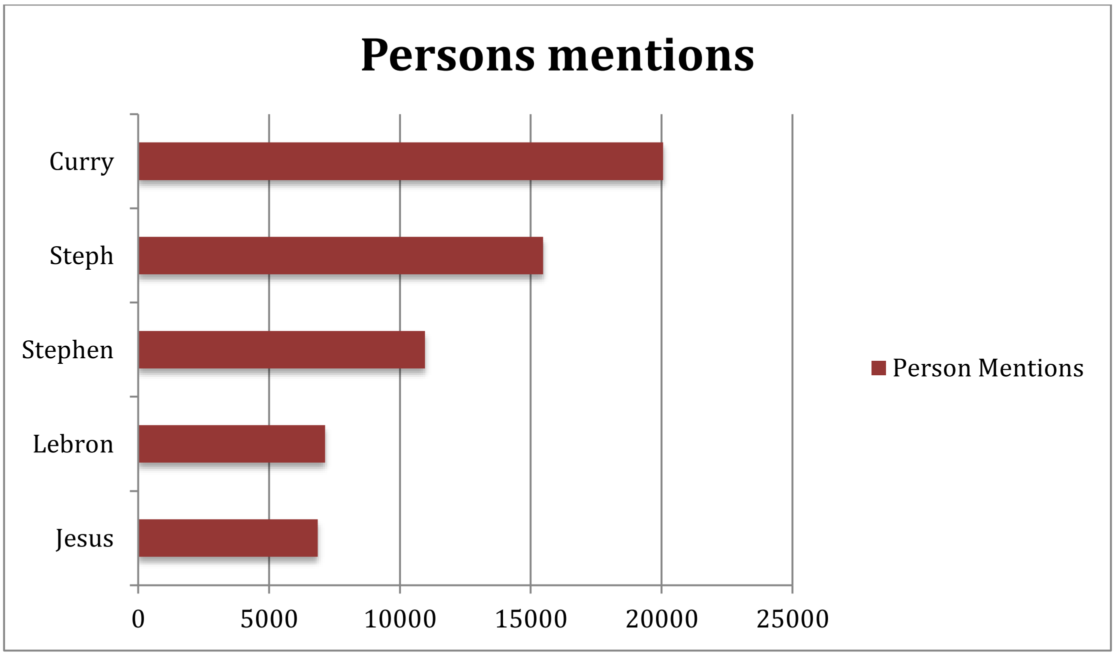


Diagram 5 - Top 5 Persons discovered based on mentions

Neo4J Queries

MATCH (m)-[r]-(n:PERSON) RETURN count(n) as person_mentions, n ORDER BY person_mentions DESC LIMIT 5

Apache Jena: Query

```
PREFIX relation_ns: <http://custom/ns/relations#>

SELECT ?entity (COUNT(?entity) as ?person_mentions)
WHERE {
        ?subject relation_ns:REFERS_TO_PERSON ?entity
}
GROUP BY ?entity
ORDER BY DESC(?person_mentions)
LIMIT 5
```

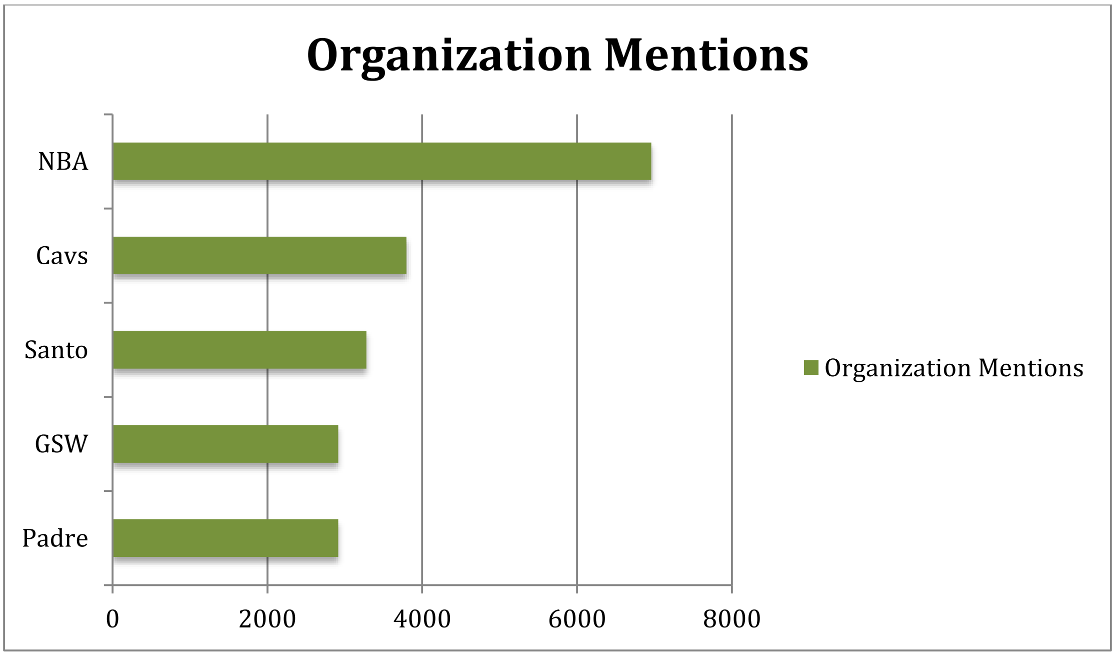


Diagram 6 - Top 5 Organizations discovered based on mentions

Neo4J Queries

```
MATCH (m)-[r]-(n:ORGANIZATION) RETURN count(n) as organization_mentions, n
ORDER BY organization_mentions DESC LIMIT 5
```

Apache Jena: Query

```
PREFIX relation_ns: <http://custom/ns/relations#>

SELECT ?entity (COUNT(?entity) as ? organization_mentions)
WHERE {
        ?subject relation_ns:REFERS_TO_ORGANIZATION ?entity
}
GROUP BY ?entity
ORDER BY DESC(?organization_mentions)
LIMIT 5
```

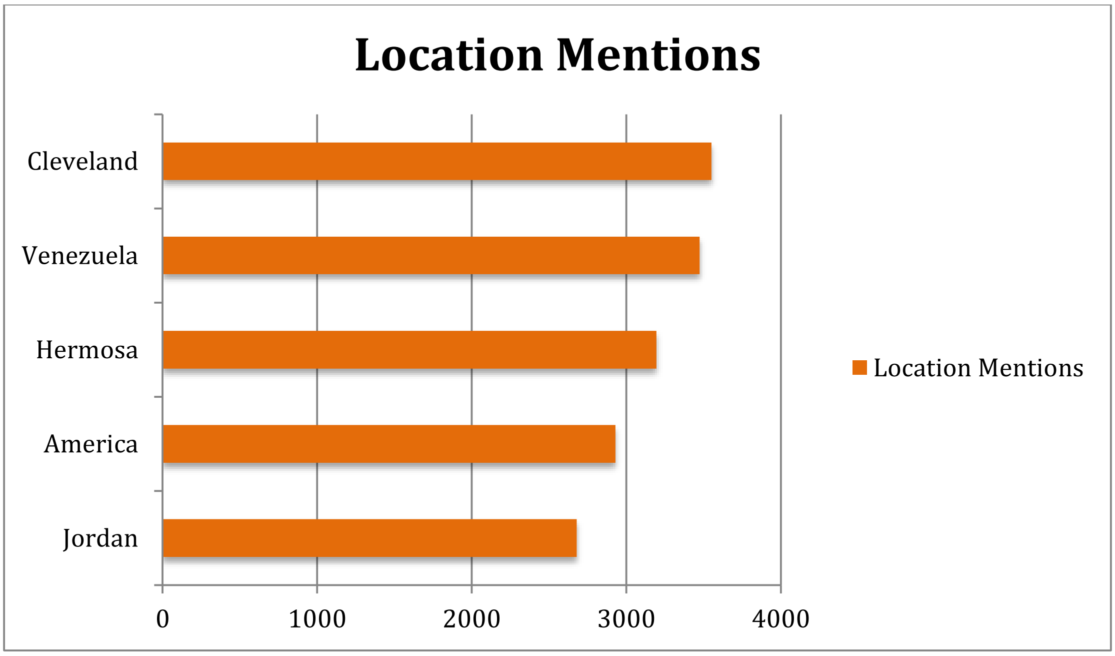


Diagram 7 - Top 5 Locations discovered based on mentions

Neo4J Queries

MATCH (m)-[r]-(n:LOCATION) RETURN count(n) as location_mentions, n ORDER BY location_mentions DESC LIMIT 5

Apache Jena: Query

```
PREFIX relation_ns: <http://custom/ns/relations#>

SELECT ?entity (COUNT(?entity) as ? location_mentions)
WHERE {
        ?subject relation_ns:REFERS_TO_LOCATION ?entity
}
GROUP BY ?entity
ORDER BY DESC(?location_mentions)
LIMIT 5
```

Queries response time

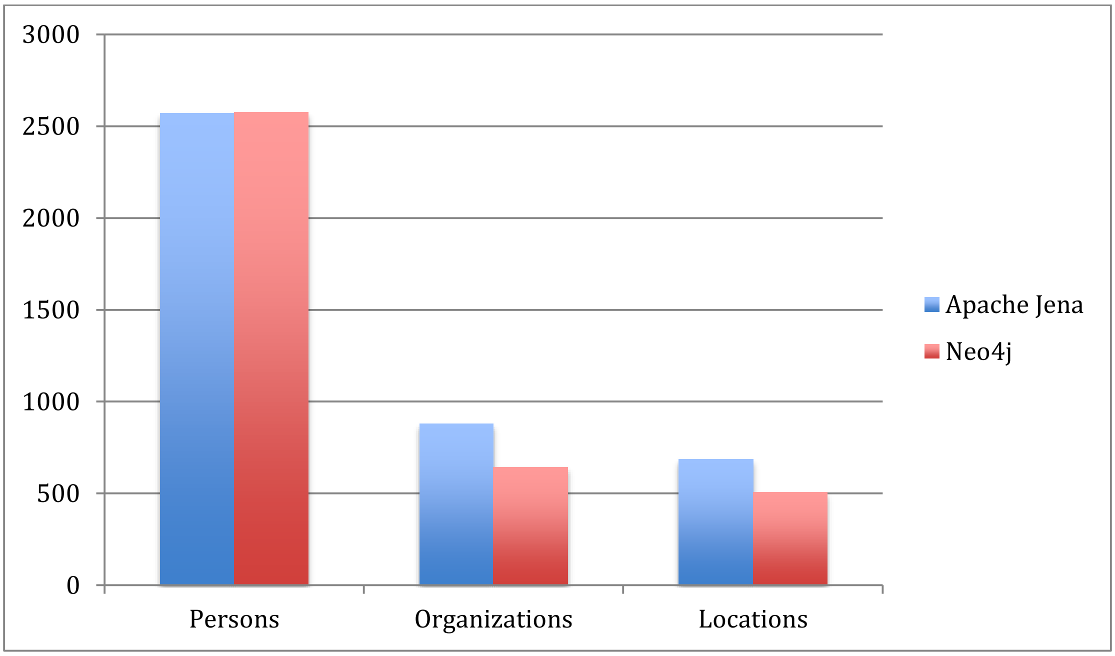


Diagram 8 - Query response time in ms

By looking the previous diagrams it seems that query time doesn't have too much difference in queries with big amount of data. In "small" queries it seems that Neo4j is a bit faster in reading, but still there is no huge time difference. The big difference between the two engines in reading is the query complexity, which is apparent if someone sees the previous queries for the two databases.

So, in real case scenarios, Neo4j can be used for sampling data fast, due to its high import time, but its ease of querying them. On the other hand, Apache Jena is very fast both in read and write, but it needs more time to build queries to display data.

Chapter 8: References

[1] Stanford [10]University. Natural Langugage Processing Group. [Online]. http://nlp.stanford.edu/software/CRF-NER.shtml

[2] Roy Thomas [1]Fielding,., 2000.

[3] R. T. - Taylor, R. N. [2]Fielding, *Principled design of the modern Web architecture.*, 2000.

[4] Silberschatz [3]Avi, *Database System Concepts, Sixth Edition.*, 2010.

[5] [4]Wikipedia. Graph database description. [Online]. https://en.wikipedia.org/wiki/Graph_database#Description

[6] W3 [5]Consortium. [Online]. https://www.w3.org/RDF/

[7] [6]Wikipedia. Wikipedia RDF image. [Online]. https://en.wikipedia.org/wiki/Resource_Description_Framework

[8] [7]Neo4j. Property graphs. [Online]. http://neo4j.com/developer/graph-database/#property-graph

[9] Gobinda G. [8]Chowdhury, "Natural Language Processing," *The Annual Review of Information Science and Technology*, 2005.

[10] Dennis Perzanowski [11]Elaine Marsh, "MUC-7 Evaluation of IE Technology: Overview of Results," April 1998.

[11] F. Rinaldi, D. Mowatt [12]W.J. Black, *Description of the NE System Used for MUC-7*.

[12] Dekang [13]Lin and Xiaoyun Wu, "Phrase clustering for discriminative learning," in *Annual Meeting of the ACL and IJCNLP*, 2009.

[13] [17]SensioLabs. Silex. [Online]. http://silex.sensiolabs.org/

[14] [9]Docker. [Online]. https://www.docker.com/what-docker

[15] [14]IETF. [Online]. https://tools.ietf.org/html/rfc5849

[16] [15]IETF. [Online]. http://tools.ietf.org/html/rfc6749

[17] [16]YAML. [Online]. http://yaml.org/

[18] [18]Fortune. [Online]. http://fortune.com/worlds-greatest-leaders/

Appendix A – Figures

Appendix B – Tables

Appendix C – Diagrams

Appendix C – Source code

https://github.com/c0nstantx/api2db