Building and Mining Knowledge Graphs Project Report

Zhangyi Wu

March 20, 2022

1 Abstract

Classical music to some extend becomes unfit for this streaming era of music, because every piece of digital classical music recording bears more complicated metadata than pop music and it is therefore harder for the streaming services to get it right. [3] The purpose of this project is to augment the way users look for classical pieces which they would like to listen. In order to reach the goal, a mini search engine specialized in classical music is built for people to search recording matching some criterion. The web GUI of the search engine includes multiple search boxes that helps users to filter the results in terms of performer, composer, duration of the recording. The search engine is grounded on knowledge graph using MusicBrainz database, and it leverages SPARQL querying techniques via endpoint located in a local GraphDB repository where all the triples are.

2 Significance

The streaming era of music is celebrated by music lovers since now people are allowed to play music they enjoy without limit if they choose to pay monthly or annually to a streaming service. Most major streaming services provide music with complete metadata, and users can use the search function to find what they want precisely and stream at once. However, the process of finding recordings could be less fulfilling for people who dig into classical music. Classical music suffer from what is not such a problem for pop music because there are various of recordings for a same piece. For example, if a user would like to listen to a Beethoven symphony which was conducted by Karajan and recorded in the 1960s, what he/she might do is typing "Beethoven", "5th Symphony" and "Karajan" in the search field and the search results could be very disordered and it can take minutes to locate the recording by the right conductor recorded in the right year. To sum up, searching classical music specifically in major music services precisely is usually a struggle which has been complained for a long while[3, 4], and this is where the impetus of this project comes from. And to improve that, this project grounded on knowledge graph extracted from Musicbrainz database provide a intuitive interface for users to interact by filling what they want to listen from which composer, performed by which musician. This project marks a different model than streaming services' built-in search functions. Although not yet equipped with semantic features, this project can serve as the first step toward a greater possibility for classical aficionados to explore.

3 Related Work

There are significant progress happening in the streaming industry. Some large music streaming services like Apple Music started to provide a composer field for some classical recordings few years ago. Some streaming services specializing in classical music emerged in recent years targeting at classical music aficionados namely Adagio or Primephonic. There are also researchers working on linking jazz performances and musicological metadata using triples[2]. An similar and exciting work by *Cota et al.* [1] explored the complex classical music information represented in RDF and even built a chatbot which can answer classical music questions. What differs my approach from theirs is the consideration for the recording rather than a composition itself.

4 Goal and Specific Objectives

The aim of this project is to build a semantic search system specialized in classical music. The user can interact with the program with input like: "Beethoven's 9th Symphony conducted in 1980s", which will be parsed by the system and will be translated to a structured SPARQL query. The query will then be executed and a list of recordings will be fetched as the result. The answer will be guaranteed by a quality metric. The development of the semantic QA system will use knowledge graph represented in RDF triples and NLP techniques like Named-entity recognition or even entity embedding. In case the NER is not performing well given unstructured data in the classical music context, the project would shift to searching recording based on given keywords related to import attributes of classical works.

5 Setup & Methodology

In this section how I obtained the RDF version of Musicbrainz and how I construct the SPARQL query for the search engine is briefly examined. The complete step-by-step setup process can be found in the appendix A.

5.1 Setup

The first step of setting the project is installing the Musicbrainz relational database following the official instruction. Musicbrainz is an open music encyclopedia that collects music metadata and the entire information of Musicbrainz is available to the public. The PostgreSQL relational database, which has 31GB in size, is the data sources for the knowledge graph I will be exploring. I use the R2RML mapping files for MusicBrainz schema and modify it a little according to the primary inclusion for classical music.

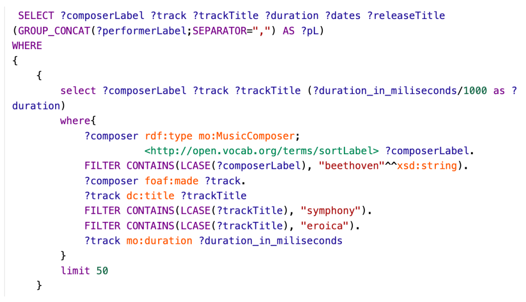
In order to save limited local disk space, I have to do the above steps in a virtual private server. I use `r2rml.jar` to generated triples and the 12 files generated triples which take up 9.01 GB space. All the triples were imported to a local GraphDB repository after being downloaded from the VPS (Due to the limitation of GraphDB which only allows RDF files less than 200MB to be uploaded, 12 RDF files were respectively split into multiple pieces of smaller RDF files before being imported). The overview of the imported RDF data can be found in the Appendix.

After all the RDF data are imported the development of search engine can start. In this project, the approach of searching classical music recording is to provide multiple fields for user to fill in appropriate information, then the backend receives all the inputs from user and construct a SPARQL query that represent what the user intends to find. The available fields for user to fill in include:

1. Track title (e.g. "piano soanta", "symphony 5", "in C minor"...)
2. Composer (e.g. "Beethoven", "Schubert"...)
3. Performer (e.g. "orchestra", "Rubinstein"...)
4. Sort the results by track duration (To satisfy the need for longer or slower version of a same musical piece)
5. Sort the results by release date
6. Limit value which constrains the number of rows of SPARQL results.

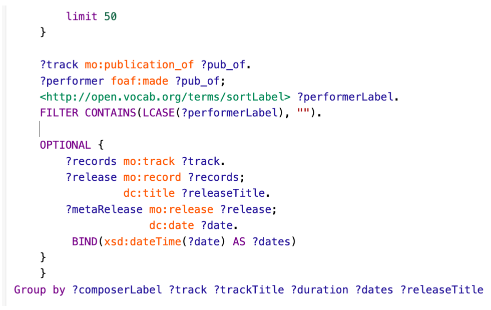
The query will be executed against GraphDB endpoint (localhost) and answer will be generated and presented to the user.

Users can explore the search engine using a website which is built with Python and is Flask-based. In the search page there are a set of text fields and a submit button for user to make input. The interaction between Flask and GraphDB endpoint is realized by python libraries including only RDFLib and SPARQLWrapper. SPARQLWrapper returns SPARQL results in CSV format and I used pandas to render it as an HTML table which is displayed in the result page.



```
 SELECT ?composerLabel ?track ?trackTitle ?duration ?dates ?releaseTitle
(GROUP_CONCAT(?performerLabel;SEPARATOR=",") AS ?pL)
WHERE
{
    {
        select ?composerLabel ?track ?trackTitle (?duration_in_miliseconds/1000 as ?
duration)
        where{
            ?composer rdf:type mo:MusicComposer;
                      <http://open.vocab.org/terms/sortLabel> ?composerLabel.
            FILTER CONTAINS(LCASE(?composerLabel), "beethoven"^^xsd:string).
            ?composer foaf:made ?track.
            ?track dc:title ?trackTitle
            FILTER CONTAINS(LCASE(?trackTitle), "symphony").
            FILTER CONTAINS(LCASE(?trackTitle), "eroica").
            ?track mo:duration ?duration_in_miliseconds
        }
        limit 50
    }
```

(a) Subquery for Finding Composer and Tracks. An example with "symphony eroica" as track title and "Beethoven" as composer



```
        limit 50
    }

    ?track mo:publication_of ?pub_of.
    ?performer foaf:made ?pub_of;
    <http://open.vocab.org/terms/sortLabel> ?performerLabel.
    FILTER CONTAINS(LCASE(?performerLabel), "").

    OPTIONAL {
        ?records mo:track ?track.
        ?release mo:record ?records;
                 dc:title ?releaseTitle.
        ?metaRelease mo:release ?release;
                     dc:date ?date.
         BIND(xsd:dateTime(?date) AS ?dates)
    }
    }
Group by ?composerLabel ?track ?trackTitle ?duration ?dates ?releaseTitle
```

(b) Query Template To Filter Performer "Karajan"

Figure 1: Screenshots of Query Template

5.2 Query Construction

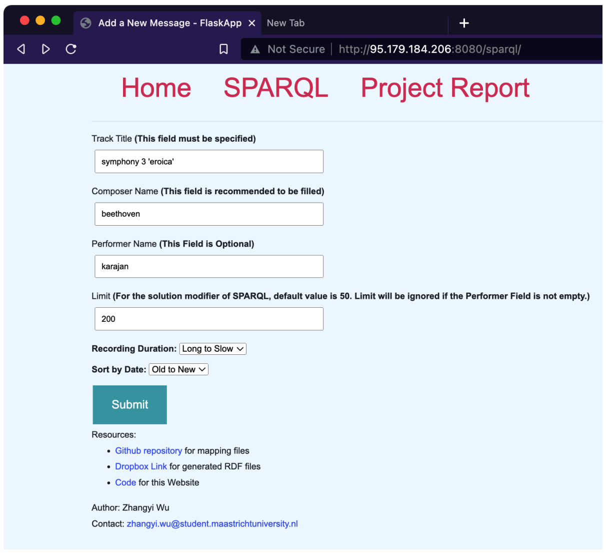
The system finds matched recording by facilitating SPARQL queries templates. Figure 1 depicts how a search instance is described in SPARQL query language.

Now let me discuss the objectives of each part of the query. It can be seen that there is an object `mo:MusicComposer` inside the subquery 1a. This type did not exist in the original generated RDF data, which only give equal treatment to performers and composers by the object `mo:MusicArtist`. This had caused hindrance to my objective, because the role of composer is a crucial thing classical music which I want to quickly identify. To tackle this issue, I insert a `rdf:type mo:MusicComposer` relation to subject `mo:MusicArtist` which were linked to a `work` by the `mo:Composition` (The query for inserting can be found in the appendix).

Since the total statements of the repository number 72,215,701, and there are 1.2641807E7 track instances in the stack (even after removing all non-classical instances already), it was be very slow to directly select tracks that meet given conditions at first. It usually took several minutes to filter tracks to only return the results whose titles contain a certain string. I started looking for how to optimize query process given million of triples in the repository and try to see what went wrong with "Explain plan and onto:measure" (Appendix C), also I tried to follow this page in the GraphDB documentation. Based on the documentation, I re-install the repository but in the configuration change `entity-index-size` to 100,000,000, which is higher than the size of unique entities in data. The query starts to execute with remarkable improvement from several seconds to around 1 second.

After some query optimization technique was investigated and I decided on putting a subquery for selecting composer and track title and set a limit for that subquery. The subquery enable me to limit the number of results from the `?track` variable to avoid timeout errors. Inside the subquery(Figure 1a), the `?composerLabel` variable is filtered to find the input composer. When the composer is not empty, it is faster starting from thousands of composers than directly filter millions of tracks. Durations of a recording in ms is also fetched with `mo:duration` predicate, and converted to seconds. With the sorting option the results can be display from longest to slowest or reversely.

In figure 1b, after the subquery is finished, the `mo:publication_of` predicate is used to find the official publication of a track, which is performed(`foaf:made`) by one or more performers (could be a pianist, conductor, or orchestra). It is not a rare case that one-to-many relation exists between a track and performers, so `GROUP_CONCAT` function is applied to `?performerLabel` in order to concatenate of all of the performers that are bound the same publication in the result. The `OPTIONAL` part is for finding the releasing date of a track. To trace back from track to date there are many intermediate predicates, but the actual reason for a `OPTIONAL` clause is that only a few `?tracks` can be linked with a `?date` data. Thus I want to keep showing relevant recording in the result table regardless of the existence of date.


Add a New Message - FlaskApp
New Tab
Not Secure | http://95.179.184.206:8080/sparql/
Home
SPARQL
Project Report

Track Title **(This field must be specified)**

symphony 3 'eroica'

Composer Name **(This field is recommended to be filled)**

beethoven

Performer Name **(This Field is Optional)**

karajan

Limit **(For the solution modifier of SPARQL, default value is 50. Limit will be ignored if the Performer Field is not empty.)**

200

Recording Duration: Long to Slow

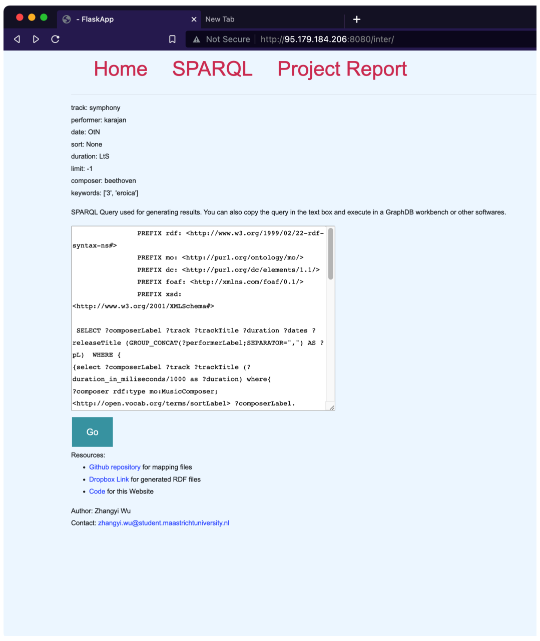
Sort by Date: Old to New

Resources:

- Github repository for mapping files
- Dropbox Link for generated RDF files
- Code for this Website

Author: Zhangyi Wu
Contact: zhangyi.wu@student.maastrichtuniversity.nl

Figure 2: Page for User to Type Inputs url:('/sparql')



track: symphony
performer: karajan
date: OtN
sort: None
duration: LtS
limit: -1
composer: beethoven
keywords: ['3', 'eroica']

SPARQL Query used for generating results. You can also copy the query in the text box and execute in a GraphDB workbench or other softwares.

```
                PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-
syntax-ns#>
                PREFIX mo: <http://purl.org/ontology/mo/>
                PREFIX dc: <http://purl.org/dc/elements/1.1/>
                PREFIX foaf: <http://xmlns.com/foaf/0.1/>
                PREFIX xsd:
<http://www.w3.org/2001/XMLSchema#>

 SELECT ?composerLabel ?track ?trackTitle ?duration ?dates ?
releaseTitle (GROUP_CONCAT(?performerLabel;SEPARATOR=",") AS ?
pL)  WHERE {
{select ?composerLabel ?track ?trackTitle (?
duration_in_miliseconds/1000 as ?duration) where{
?composer rdf:type mo:MusicComposer;
<http://open.vocab.org/terms/sortLabel> ?composerLabel.
```

Resources:

- Github repository for mapping files
- Dropbox Link for generated RDF files
- Code for this Website

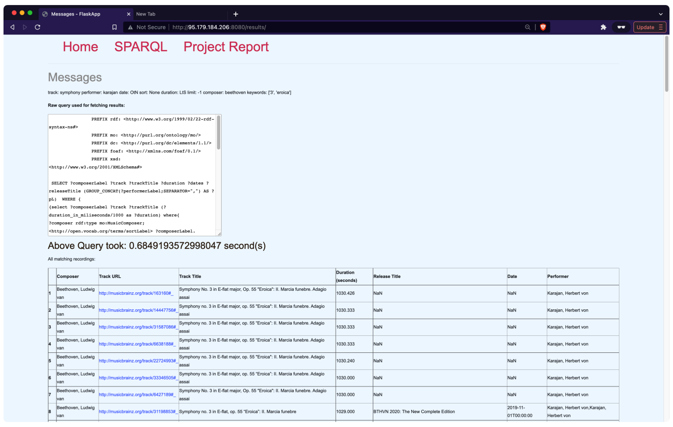
Author: Zhangyi Wu
Contact: zhangyi.wu@student.maastrichtuniversity.nl

Figure 3: Page Showing Query Information url:('/interm')

6 Results

The visible result of this project is basically a webpage that carries out classical music searching. Figure 2 shows the simple implemented website that receives input from users. In the example, the user provides some parameters namely Track Title ("symphony 3"), Composer Name ("beethoven"), Performer Name ("karajan"), Limit 200 instances to be shown, the durations are sorted from long to slow in the results. After a user hits the "submit" button, the user will be redirected to a intermediate page (Figure 3) where there is a textarea showing a constructed SPARQL query which is parsed from what the user just filled in and will be executed later on. Now the user can see what results the query fetches by hitting the "Go" button. The result page list all the matched results and the performance of the query, which as indicated took 0.684 seconds to execute.

Figure 5 shows another example of looking for the 9th Nocturne composed by Chopin. And the results are sorted from oldest to newest in terms of release date (if release date is not empty).



Messages

track: symphony performer: karajan date: OtN sort: None duration: LtS limit: -1 composer: beethoven keywords: ['3', 'eroica']

Raw query used for fetching results:

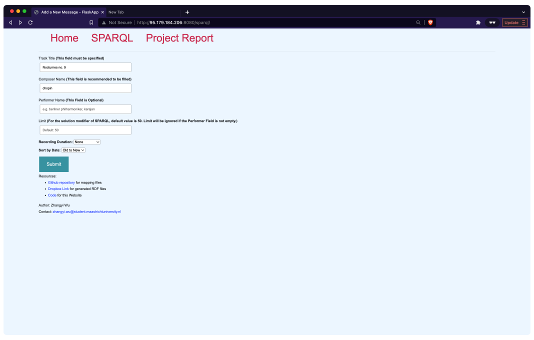
```
                PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-
syntax-ns#>
                PREFIX mo: <http://purl.org/ontology/mo/>
                PREFIX dc: <http://purl.org/dc/elements/1.1/>
                PREFIX foaf: <http://xmlns.com/foaf/0.1/>
                PREFIX xsd:
<http://www.w3.org/2001/XMLSchema#>

 SELECT ?composerLabel ?track ?trackTitle ?duration ?dates ?
releaseTitle (GROUP_CONCAT(?performerLabel;SEPARATOR=",") AS ?
pL)  WHERE {
{select ?composerLabel ?track ?trackTitle (?
duration_in_miliseconds/1000 as ?duration) where{
?composer rdf:type mo:MusicComposer;
<http://open.vocab.org/terms/sortLabel> ?composerLabel.
```

Above Query took: 0.6849193572998047 second(s)

All matching recordings:

	Composer	Track URL	Track Title	Duration (seconds)	Release Title	Date	Performer
1	Beethoven, Ludwig van	http://musicbrainz.org/track/163160#_	Symphony No. 3 in E-flat major, Op. 55 "Eroica": II. Marcia funebre. Adagio assai	1030.426	NaN	NaN	Karajan, Herbert von
2	Beethoven, Ludwig van	http://musicbrainz.org/track/14447756#_	Symphony no. 3 in E-flat major, op. 55 "Eroica": II. Marcia funebre. Adagio assai	1030.333	NaN	NaN	Karajan, Herbert von
3	Beethoven, Ludwig van	http://musicbrainz.org/track/31587086#_	Symphony no. 3 in E-flat major, op. 55 "Eroica": II. Marcia funebre. Adagio assai	1030.333	NaN	NaN	Karajan, Herbert von
4	Beethoven, Ludwig van	http://musicbrainz.org/track/6638188#_	Symphony no. 3 in E-flat major, op. 55 "Eroica": II. Marcia funebre. Adagio assai	1030.333	NaN	NaN	Karajan, Herbert von
5	Beethoven, Ludwig van	http://musicbrainz.org/track/22724993#_	Symphony no. 3 in E-flat major, op. 55 "Eroica": II. Marcia funebre. Adagio assai	1030.240	NaN	NaN	Karajan, Herbert von
6	Beethoven, Ludwig van	http://musicbrainz.org/track/33346505#_	Symphony no. 3 in E-flat major, op. 55 "Eroica": II. Marcia funebre. Adagio assai	1030.000	NaN	NaN	Karajan, Herbert von
7	Beethoven, Ludwig van	http://musicbrainz.org/track/6427189#_	Symphony No. 3 in E-flat major, Op. 55 "Eroica": II. Marcia funebre. Adagio assai	1030.000	NaN	NaN	Karajan, Herbert von
8	Beethoven, Ludwig van	http://musicbrainz.org/track/31198853#_	Symphony no. 3 in E-flat, op. 55 "Eroica": II. Marcia funebre	1029.000	BTHVN 2020: The New Complete Edition	2019-11-01T00:00:00	Karajan, Herbert von,Karajan, Herbert von

Figure 4: Page Showing Multi-column Results . url:(/results)


Add a New Message - FlaskApp
New Tab
Not Secure
http://95.179.184.206:8080/sparql/
Update
Home
SPARQL
Project Report
Submit

Track Title **(This field must be specified)**

Noctumes no. 9

Composer Name **(This field is recommended to be filled)**

chopin

Performer Name **(This Field is Optional)**

e.g. berliner philharmoniker, karajan

Limit **(For the solution modifier of SPARQL, default value is 50. Limit will be ignored if the Performer Field is not empty.)**

Default: 50

Recording Duration: None

Sort by Date: Old to New

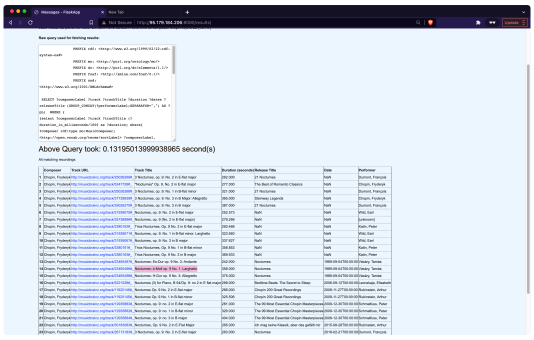
Resources:

- Github repository for mapping files
- Dropbox Link for generated RDF files
- Code for this Website

Author: Zhangyi Wu

Contact: zhangyi.wu@student.maastrichtuniversity.nl

(a) An example with "Nocturnes no. 9" as track title and "chopin" as composer



Raw query used for fetching results:

```
                PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-
syntax-ns#>
                PREFIX mo: <http://purl.org/ontology/mo/>
                PREFIX dc: <http://purl.org/dc/elements/1.1/>
                PREFIX foaf: <http://xmlns.com/foaf/0.1/>
                PREFIX xsd:
<http://www.w3.org/2001/XMLSchema#>

 SELECT ?composerLabel ?track ?trackTitle ?duration ?dates ?
releaseTitle (GROUP_CONCAT(?performerLabel;SEPARATOR=",") AS ?
pL)  WHERE {
{select ?composerLabel ?track ?trackTitle (?
duration_in_miliseconds/1000 as ?duration) where{
?composer rdf:type mo:MusicComposer;
<http://open.vocab.org/terms/sortLabel> ?composerLabel.
```

Above Query took: 0.13195013999938965 second(s)

All matching recordings:

	Composer	Track URL	Track Title	Duration (seconds)	Release Title	Date	Performer
1	Chopin, Fryderyk	http://musicbrainz.org/track/25026269#_	3 Nocturnes, op. 9: No. 2 in E-flat major	262.000	21 Nocturnes	NaN	Dumont, François
2	Chopin, Fryderyk	http://musicbrainz.org/track/5247726#_	"Nocturnes" Op. 9, No. 2 in E-flat major	277.000	The Best of Romantic Classics	NaN	Chopin, Fryderyk
3	Chopin, Fryderyk	http://musicbrainz.org/track/25026268#_	3 Nocturnes, op. 9: No. 1 in B-flat minor	321.000	21 Nocturnes	NaN	Dumont, François
4	Chopin, Fryderyk	http://musicbrainz.org/track/37728839#_	3 Nocturnes, op. 9: No. 3 in B Major: Allegretto	365.000	Steinway Legends	NaN	Chopin, Fryderyk
5	Chopin, Fryderyk	http://musicbrainz.org/track/25026270#_	3 Nocturnes, op. 9: No. 3 in B major	387.000	21 Nocturnes	NaN	Dumont, François
6	Chopin, Fryderyk	http://musicbrainz.org/track/31939075#_	Nocturnes, op. 9: No. 2 in E-flat major	252.573	NaN	NaN	Wild, Earl
7	Chopin, Fryderyk	http://musicbrainz.org/track/35738996#_	Nocturnes, op. 9 (No. 2 in E-flat major)	279.266	NaN	NaN	[unknown]
8	Chopin, Fryderyk	http://musicbrainz.org/track/3380162#_	Trios Nocturnes, Op. 9 No. 2 in E-flat major	293.466	NaN	NaN	Katin, Peter
9	Chopin, Fryderyk	http://musicbrainz.org/track/31939071#_	Nocturnes, op. 9: No. 1 in B-flat minor. Larghetto	323.560	NaN	NaN	Wild, Earl
10	Chopin, Fryderyk	http://musicbrainz.org/track/31939067#_	Nocturnes, op. 9: No. 3 in B major	337.627	NaN	NaN	Wild, Earl
11	Chopin, Fryderyk	http://musicbrainz.org/track/3380161#_	Trios Nocturnes, Op. 9 No. 1 in B-flat minor	358.853	NaN	NaN	Katin, Peter
12	Chopin, Fryderyk	http://musicbrainz.org/track/3380163#_	Trios Nocturnes, Op. 9 No. 3 in B major	369.933	NaN	NaN	Katin, Peter
13	Chopin, Fryderyk	http://musicbrainz.org/track/23465497#_	Nocturnes: Es-Dur op. 9 No. 2: Andante	242.000	Nocturnes	1989-09-04T00:00:00	Vásáry, Tamás
14	Chopin, Fryderyk	http://musicbrainz.org/track/23465496#_	Nocturnes: b-Moll op. 9 No. 1: Larghetto	358.000	Nocturnes	1989-09-04T00:00:00	Vásáry, Tamás
15	Chopin, Fryderyk	http://musicbrainz.org/track/23465498#_	Nocturnes: H-Dur op. 9 No. 3: Allegretto	375.000	Nocturnes	1989-09-04T00:00:00	Vásáry, Tamás
16	Chopin, Fryderyk	http://musicbrainz.org/track/5221528#_	Nocturnes (3) for Piano, B 54/Op. 9: no 2 in E flat major	299.000	Bedtime Beats: The Secret to Sleep	2006-09-12T00:00:00	Leonskaja, Elisabeth
17	Chopin, Fryderyk	http://musicbrainz.org/track/11920146#_	Nocturnes Op. 9 No. 2 in E-flat major	266.000	Chopin 200 Great Recordings	2009-11-27T00:00:00	Rubinstein, Arthur
18	Chopin, Fryderyk	http://musicbrainz.org/track/11920145#_	Nocturnes Op. 9 No. 1 in B-flat minor	325.506	Chopin 200 Great Recordings	2009-11-27T00:00:00	Rubinstein, Arthur
19	Chopin, Fryderyk	http://musicbrainz.org/track/12935883#_	Nocturnes, op. 9: no. 2 in E-flat major	281.000	The 99 Most Essential Chopin Masterpieces	2009-12-30T00:00:00	Schmalfuss, Peter
20	Chopin, Fryderyk	http://musicbrainz.org/track/12935882#_	Nocturnes, op. 9: no. 1 in B-flat minor	326.000	The 99 Most Essential Chopin Masterpieces	2009-12-30T00:00:00	Schmalfuss, Peter
21	Chopin, Fryderyk	http://musicbrainz.org/track/12935884#_	Nocturnes, op. 9: no. 3 in B major	404.000	The 99 Most Essential Chopin Masterpieces	2009-12-30T00:00:00	Schmalfuss, Peter
22	Chopin, Fryderyk	http://musicbrainz.org/track/30183083#_	Nocturnes, Op. 9 No. 2 in E-Flat Major	265.000	Ich mag keine Klassik, aber das gefällt mir	2015-08-28T00:00:00	Rubinstein, Arthur
23	Chopin, Fryderyk	http://musicbrainz.org/track/26713183#_	3 Nocturnes, op. 9: No. 2 in E-flat major	263.000	Nocturnes	2018-02-21T00:00:00	Dumont, François

(b) Results for "Nocturnes no. 9" composed by Chopin

Figure 5: An Instance of Searching a Nocturne by Chopin

7 Discussion

In the beginning I came across a lot of query timeout, I managed to to prevent it by resorting to subquery, but the result is not significantly improved. I finally find the right way to do it, which is to re-configure the repository by changing the entity size, and it proved to make huge difference in terms of query speed. The shortcoming of using a subquery is that the subquery selects a limited amount of instances, so if there are 100,000 tracks that contain "symphony 3" in their titles, and the subquery only select 500 of them with a `LIMIT` clause. The following step, which is filtering performers of selected tracks, has to be done with incomplete track instances. In other words, the remaining query is selecting from the 500 candidate tracks rather than 100,000 tracks. To fix that, I choose to ignore the limit when performer is specified. Also, in figure 1b an "OPTIONAL" clause was declared due to the incomplete information of dates. What I suspect to be the reason is either MusicBrainz's data is imcomplete, or is due to some mistakes in the mapping files, for example the relationship between tables is not well established.

What is remained unexplored, however, is the semantic parsing of user input. I have attempted to receive semi natural language as input like "beethoven's 5th symphony" and capture the key elements that can be filled in the slots of the template query using Named-entity recognition. The NLP pipeline using small English model (en_core_web_sm) did not show competence for identifying the important entity input above as what I expected. The word "Beethoven" can be categorized as ORG or a PERSON randomly. And due to the limited time for this project I could not work out an ideal method to identifying entity with acceptable accuracy, either have enough time to find relevant musical corpus and train a custom NER model. It forces me to change my proposed plan and fall back to search by keywords including composer, title etc.

8 Conclusions

The proposed system implements a mini search engine for RDF data in the context of classical music. The front-end of the application receives inputs from user and replaces query template with keywords, executes the query via a SPARQL interface and retrieved results that match the input condition. In the initial phrase, it took a long time for a query to be executed, which obviously does a disservice to the experiments of using this search engine, but this is well solved after the entity index size is changed. The search system captures the input from user and replace the corresponding elements in the SPARQL query to find recordings that match user's preference.

The limitation of the system is first, the data is not very complete in terms of releasing date. There are also more attribute of music metadata this application can support but have not yet implemented like the nationality (area) of composer, so there are still a lot of possible advantages of Musicbrainz database to be taken.

The semantic feature is not realized as expected in the proposal. As for the further developments, the actual semantic search is of course an exciting topic to explore to enhance the current system. I do think this is a very exciting for researching because the current NER and other NLP tools can really make textual analysis of unstructured input easier for algorithm to understand. Once it is realized, I believe it will earn much acclaim from various of crowds who love classical music.

Appendix A Setup Guide

In this section of appendix I will list every step I took to set up this project. Since I do not have enough space in my computer, some steps were performed in a VPS server running Ubuntu OS (version 22.04). The local part depends on GraphDB and various python modules (listed in the requirement.txt). I was running the application in a Python 3.9 environment. To reproduce the work, simple start from step 4 by downloading RDF files uploaded to my Dropbox folder with the highlighed link, and import each TURTLE files to a GraphDB repository.

1. (On the server) Download mbdump from this page
2. (On the server) Install Musicbrainz Database following the instruction
3. (On the server) Clone the mapping files from this Github repository, and run `java -jar r2rml.jar <mappingFile>.properties` to generate 14 RDF files in TURTLE format.
4. Step 3 generates 9.01 GB of outputs files, which were first split into blocks less than 200 MB in order for GraphDB to import. The split files were uploaded to this Dropbox directory
5. Import all the RDF data to a repository in GraphDB. Also some namespaces have to be added:

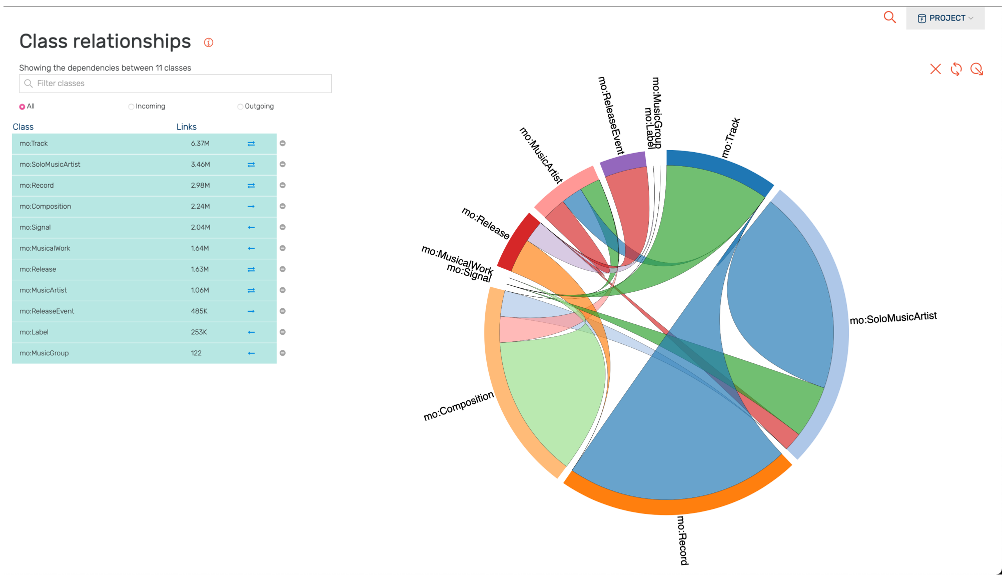
dc	http://purl.org/dc/elements/1.1/
foaf	http://xmlns.com/foaf/0.1/
gn	http://www.geonames.org/ontology#
mo	http://purl.org/ontology/mo/
owl	http://www.w3.org/2002/07/owl#
path	http://www.ontotext.com/path#
rdf	http://www.w3.org/1999/02/22-rdf-syntax-ns#
rdfs	http://www.w3.org/2000/01/rdf-schema#
wgs	http://www.w3.org/2003/01/geo/wgs84_pos#
xsd	http://www.w3.org/2001/XMLSchema#

   After adding those namespaces, the "mo:MusicComposer" rule should be inserted for mo:MusicArtist who has compositions. The application could not be working without inserting that relation. The query for inserting is: `insert ?composer rdf:type mo:MusicComposer WHERE ?work rdf:type mo:Composition. ?work mo:composer ?composer. ?composer rdf:type mo:MusicArtist.`

6. Building website and query templates.
7. To run the Flask application, go to directory `bmkg_project` in the command line, and run this two commands: `export FLASK_APP=server` and `flask run` and the project should be running on http://127.0.0.1:5000/

Appendix B Overview of RDF

The graph below showing the top relationships between classes is a screenshot from GraphDB. In the left a list of all classes order by number of links they have. There are millions of links for some most populous classes. In the right is a visualization displaying links between the individual instances of two classes. It is easy to observe how strong two classes are connected.



Class relationships

Showing the dependencies between 11 classes

Filter classes

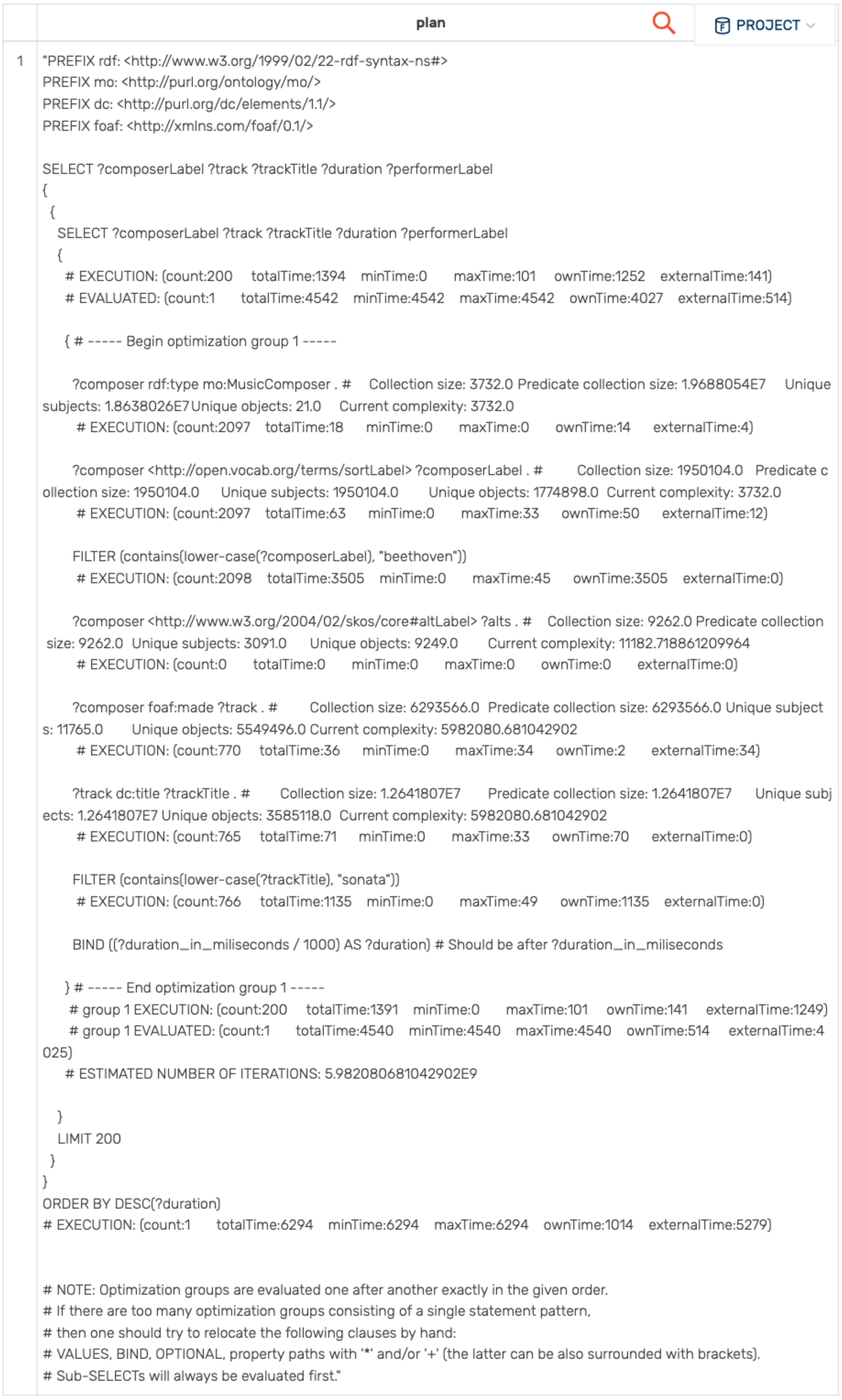
[✓ All] Incoming Outgoing

Class	Links	
mo:Track	6.37M	⇄
mo:SoloMusicArtist	3.46M	⇄
mo:Record	2.98M	⇄
mo:Composition	2.24M	→
mo:Signal	2.04M	←
mo:MusicalWork	1.64M	←
mo:Release	1.63M	⇄
mo:MusicArtist	1.06M	⇄
mo:ReleaseEvent	485K	→
mo:Label	253K	←
mo:MusicGroup	122	←

Figure 6: Overview of Class Relationships

Appendix C Execution Process

Following an advice on this website. I run the query with `FROM onto:measure` clause in order to find out how the engine would evaluate my query, and see the timings for each separate step of the execution process. I also use this as a reference to optimize my query. Here is the result from the explain plan:



```
"PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX mo: <http://purl.org/ontology/mo/>
PREFIX dc: <http://purl.org/dc/elements/1.1/>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>

SELECT ?composerLabel ?track ?trackTitle ?duration ?performerLabel
{
 {
  SELECT ?composerLabel ?track ?trackTitle ?duration ?performerLabel
  {
   # EXECUTION: (count:200   totalTime:1394   minTime:0     maxTime:101   ownTime:1252   externalTime:141)
   # EVALUATED: (count:1     totalTime:4542   minTime:4542   maxTime:4542   ownTime:4027   externalTime:514)

   { # ----- Begin optimization group 1 -----

    ?composer rdf:type mo:MusicComposer . #    Collection size: 3732.0 Predicate collection size: 1.9688054E7    Unique
subjects: 1.8638026E7 Unique objects: 21.0    Current complexity: 3732.0
    # EXECUTION: (count:2097   totalTime:18     minTime:0     maxTime:0     ownTime:14     externalTime:4)

    ?composer <http://open.vocab.org/terms/sortLabel> ?composerLabel . #      Collection size: 1950104.0   Predicate c
ollection size: 1950104.0    Unique subjects: 1950104.0    Unique objects: 1774898.0  Current complexity: 3732.0
    # EXECUTION: (count:2097   totalTime:63     minTime:0     maxTime:33    ownTime:50     externalTime:12)

    FILTER (contains(lower-case(?composerLabel), "beethoven"))
    # EXECUTION: (count:2098   totalTime:3505   minTime:0     maxTime:45    ownTime:3505   externalTime:0)

    ?composer <http://www.w3.org/2004/02/skos/core#altLabel> ?alts . #   Collection size: 9262.0 Predicate collection
 size: 9262.0  Unique subjects: 3091.0    Unique objects: 9249.0      Current complexity: 11182.718861209964
    # EXECUTION: (count:0      totalTime:0      minTime:0     maxTime:0     ownTime:0      externalTime:0)

    ?composer foaf:made ?track . #      Collection size: 6293566.0  Predicate collection size: 6293566.0 Unique subject
s: 11765.0     Unique objects: 5549496.0 Current complexity: 5982080.681042902
    # EXECUTION: (count:770    totalTime:36     minTime:0     maxTime:34    ownTime:2      externalTime:34)

    ?track dc:title ?trackTitle . #      Collection size: 1.2641807E7      Predicate collection size: 1.2641807E7     Unique subj
ects: 1.2641807E7 Unique objects: 3585118.0  Current complexity: 5982080.681042902
    # EXECUTION: (count:765    totalTime:71     minTime:0     maxTime:33    ownTime:70     externalTime:0)

    FILTER (contains(lower-case(?trackTitle), "sonata"))
    # EXECUTION: (count:766    totalTime:1135   minTime:0     maxTime:49    ownTime:1135   externalTime:0)

    BIND ((?duration_in_miliseconds / 1000) AS ?duration) # Should be after ?duration_in_miliseconds

   } # ----- End optimization group 1 -----
    # group 1 EXECUTION: (count:200    totalTime:1391   minTime:0     maxTime:101    ownTime:141    externalTime:1249)
    # group 1 EVALUATED: (count:1     totalTime:4540   minTime:4540   maxTime:4540   ownTime:514    externalTime:4
025)
   # ESTIMATED NUMBER OF ITERATIONS: 5.982080681042902E9

  }
  LIMIT 200
 }
}
ORDER BY DESC(?duration)
# EXECUTION: (count:1     totalTime:6294   minTime:6294   maxTime:6294   ownTime:1014   externalTime:5279)


# NOTE: Optimization groups are evaluated one after another exactly in the given order.
# If there are too many optimization groups consisting of a single statement pattern,
# then one should try to relocate the following clauses by hand:
# VALUES, BIND, OPTIONAL, property paths with '*' and/or '+' (the latter can be also surrounded with brackets).
# Sub-SELECTs will always be evaluated first."
```

Figure 7: Outputs of Executing: `SELECT * FROM onto:measure`

References

[1] G Cota et al. Representing complex knowledge for exploration and recommendation: The case of classical music information. *Applications and Practices in Ontology Design, Extraction, and Reasoning*, 49:107, 2020.

[2] Terhi Nurmikko-Fuller, Daniel Bangert, Yun Hao, and J. Stephen Downie. Swinging triples: Bridging jazz performance datasets using linked data. In *Proceedings of the 1st International Workshop on Semantic Applications for Audio and Music*, SAAM '18, page 42–45, New York, NY, USA, 2018. Association for Computing Machinery.

[3] Jamie Powell. Classical music has a metadata problem, Jun 2019.

[4] Ben Sisario. In streaming age, classical music gets lost in the metadata, Jun 2019.