Universidade de São Paulo

Instituto de Ciências Matemáticas e de Computação

GEO Database and API Manual

Author
André Moreira Souza

Supervisor
Prof. Elaine Parros Machado de Sousa

July 3, 2020

GEO Database and API Manual

André Moreira Souza

September 2019

Contents

Acronyms

API Application Programming Interface. 1, 3–7

CK Check. 4

CLI Command Line Interface. 5, 7

DBMS Database Management System. 3

FK Foreign Key. 4

GEO *Grupo de Estudos Olímpicos* EEFE-USP. 1, 3

GIN Generalized Inverted Index. 3, 4

HTTP HyperText Transfer Protocol. 6, 7

JSON JavaScript Object Notation. 3, 5–7

NLTK Natural Language Toolkit. 5, 6

NN Not Null. 4

NoSQL Not-only SQL. 3

PK Primary Key. 4

REST Representational State Transfer. 6, 7

URI Uniform Resource Identifier. 6

1 Introduction

This document describes the modeling and implementation of a structure for storage of data of Brazilian Olympic athletes, with emphasis on the storage and efficient recovery of interviews performed by the *Grupo de Estudos Olímpicos* EEFE-USP (GEO), as well as metadata for text mining. The structure consists of a database, implemented using advanced functionalities of the PostgreSQL 11, and a Web Application Programming Interface (API), using the Flask microframework in the Python 3 language.

2 Database

The database must be able to efficiently store and retrieve both structured data, i.e., objective data about the Brazilian Olympic athletes, Olympic editions, etc., and unstructured data, i.e., the interviews with those athletes and metadata (e.g. bag-of-words) extracted for text mining algorithms. With those goals in mind, a comparison of both relational and NoSQL DBMSs led to the choice of PostgreSQL for the implementation of this project.

The choice of PostgreSQL was mainly due to its support to the JSON datatype and full-text search techniques. The object-relational data model for storing the structured data along with these functionalities for storing the unstructured data satisfied the previous requirements, giving PostgreSQL a vantage over other evaluated DBMSs, such as MongoDB and Cassandra.

2.1 Tables

The modelling of the tables related to the interviews can be seen in figures **??** and **??**. This schema is independent of any other tables related to the objective data (Olympic editions, sports, medals, etc.), although future functionalities can be implemented once these tables are available. The created tables and their columns can be found in table 1

The table "interviews" contains the interview of each athlete, separated in three columns (the entire text, only the questions, and only the answers), the extracted metadata and columns related to full-text search indexes. To guarantee the consistency between the columns of this table, a function was created to correctly preprocess and insert an interview in the database, with details in section 2.4.

The "athletes" table contains the ID for each athlete in the database and other data related to them.

The "api_user" table contains the data from registered users for the developed API. The users present in this table (e.g. data miners) have access to restricted data, such as the interviews.

2.2 Indexes & Full-Text Search

For an efficient recovery of data in a great volume of text, PostgreSQL provides two new data types: *tsvector* and *tsquery*, that can be used respectively to preprocess and query in a text document [5].

A *tsvector* attribute consists of a structure similar to a bag-of-words of a text document, where for each stemmed token there is a list of positions where that token appears in the given text. This attribute can be indexed by a Generalized Inverted Index (GIN) [6], and queried using the full text search operators and functions provided by the DBMS [7].

In the table "interviews", as displayed in figure **??**, there are three *tsvector* attributes: one for each text / text-array column. Using the standard dictionary for the Portuguese language, each one of these attributes is generated by trigger functions before insert or update operations, and has a Generalized Inverted Index to enhance the performance on queries on the base columns. Each one of these indexes can be found in table 2.

Table 1: List of created tables, with their respective columns and descriptions.

Table	Column	Datatype	Constraints	Description
api_users	username	varchar(30)	PK, CK	Username. Uses C identifier naming rules.
	password	varchar(255)	NN	Password for the respective user. Encryption managed by the API.
athletes	id	integer	PK, CK	Athlete ID number. Must be a positive integer.
interviews	id	integer	PK, FK	Athlete ID number. References “athletes”.“id”.
	text	text	NN	Full text of the interview.
	questions	text[]	NN	Array of paragraphs spoken by the interviewer.
	answers	text[]	NN	Array of paragraphs spoken by the interviewee.
	meta	jsonb	NN	Metadata for the interview. Contains the bag-of-words, stemmed or not, for each of the previous text columns. Also contains named entities, extracted with algorithms implemented in SpaCy.
	tstext	tsvector	NN	Full text search data for indexing of queries on the “text” column.
	tsquestions	tsvector	NN	Full text search data for indexing of queries on the “questions” column.
	tsanswers	tsvector	NN	Full text search data for indexing of queries on the “answers” column.

Table 2: List of created indexes, omitting primary-key indexes.

Table	Index Name	Type	Search Key	Description
interviews	interviews_idx_tstext	GIN	tstext	Text-search index for text column.
	interviews_idx_tsquestions	GIN	tsquestions	Text-search index for questions column.
	interviews_idx_tsanswers	GIN	tsanswers	Text-search index for answers column.

2.3 Triggers

Table 3: List of created triggers.

Table	Trigger Name	Trigger Function
interviews	interviews_trigg_tstext	tsvector_update_trigger
	interviews_trigg_textarrays	interviews_textarrays_update_trigger

As mentioned in the section 2.2, triggers were created for generating the *tsvector* columns before an insert or update operation. For the "text" column, the function "*to_tsvector*" was used for this task. However, for the "questions" and "answers" columns, due to being of the text-array data type, no standard function was able to fulfill the same task, and the trigger function "*interviews_textarrays_update_trigger*" was created. A list of the created triggers can be seem in table 3.

PostgreSQL provides standard dictionaries for various languages, including Portuguese, which can be used to correctly generate the *tsvectors*.

2.4 Procedures and Inserting an Interview

Table 4: List of created functions.

Function	Parameter Name	Datatype	Description
interview_insert	id	integer	Athlete ID for the new interview.
	docx	text	Interview text in JSON format, converted from docx with *docx2json*.

Initially, the insertions and metadata generation for the "interviews" table were operated by a Python script, with the assistance of the Natural Language Toolkit (NLTK) [10] and *docx2json* [11] libraries.

Due to the support of PL/Python, this script was integrated into the database as a procedure, called "interview_insert" (see table 4) that takes a document, converted to JSON by *docx2json*, generates the metadata and inserts a new row into the "interviews" table. This adaptation leads to a smaller need for external files, giving a larger autonomy to the database, at the cost of having new dependencies (Python 3, NLTK and *docx2json*) on the same system where the PostgreSQL server is operating.

Additionally, a Python script has been created, as a Command Line Interface (CLI), to insert new interviews to the database. The CLI takes arguments for connection parameters to the database.

2.5 Users

Besides the default "postgres" user, the users "api" and "api_admin" were created for usage by the API. The "api" user has select grants for the "interviews" table, and select / update grants for the "api_users" table. The "api_admin' user has "all" grants on all the created tables. Due to the support of PL/Python, this script was integrated into the database as a procedure, called "interview_insert" that takes a document, converted to JSON by *docx2json*, generates the metadata and inserts a new row into the "interviews" table. This adaptation leads to a smaller need for external files, giving a larger autonomy to the database, at the cost of having new dependencies.

3 Web API

To facilitate the access and processing of the stored data, there must be an interface between the database and the user. This interface must be independent of the programming language used by the user, and it must also be simple and efficient, producing as little overhead as possible for accessing the desired data.

For those purposes, a Web API was modelled and implemented, using a Representational State Transfer (REST) architecture. In this manner, all data can be accessed with HTTP requests, with responses in JSON format.

3.1 Technologies Used

The RESTful API was implemented in Python 3, using the Flask micro-framework. This choice was due to Flask's simplicity, which makes the API more efficient, generating a smaller overhead in the process of accessing the data. Some extensions of Flask were also used to support the RESTful architecture and authentication methods. The *psycopg* package was used to coordinate all communication between the API and the database.

3.2 Dependencies

The following is a list of dependencies, in Python 3, required by the API:

- Flask [8]
- Flask-HTTPAuth [4]
- Flask-RESTful [2]
- NLTK [10]
- Passlib [1]
- Psycopg2 [12]
- Werkzeug [9]

3.3 Resources

Table 5: List of resources for the API.

Resource Name	Methods	URI
InterviewAll	GET	/interviews/all
InterviewAllText	GET	/interviews/all/text
InterviewAllQuestions	GET	/interviews/all/questions
InterviewAllAnswers	GET	/interviews/all/answers
InterviewAllMeta	GET	/interviews/all/meta
InterviewAny	GET	/interviews/<int_list:ids>
InterviewAnyText	GET	/interviews/<int_list:ids>/text
InterviewAnyQuestions	GET	/interviews/<int_list:ids>/questions
InterviewAnyAnswers	GET	/interviews/<int_list:ids>/answers
InterviewAnyMeta	GET	/interviews/<int_list:ids>/meta
InterviewSearch	GET	/interviews/<string:search_string>
InterviewSearchText	GET	/interviews/<string:search_string>/text
InterviewSearchQuestions	GET	/interviews/<string:search_string>/questions
InterviewSearchAnswers	GET	/interviews/<string:search_string>/answers
InterviewSearchMeta	GET	/interviews/<string:search_string>/meta
User	POST, PUT, DELETE	/users

As per the REST architecture, the API is composed of multiple independent resources, where each one has one or more HTTP access methods (GET, POST, PUT, DELETE, ...). In table 5 there are the definitions for all the resources of the API.

The user can access the entire data from the "interviews" table, as well as each column, with a HTTP request. It is also possible to filter the results with a list of athlete IDs, or with a search string.

All the resources related to the interviews are restricted to users present in the table "api_users" from the database. Every request to these resources requires a basic HTTP authentication from the API user, in its "Authorization" header.

The authentication is managed by the "User" resource, where the administrator (a database user with the required grants on the "api_users" table) can create and delete users from the API, and users can change their passwords. For requests to the "User" resource, the authentication data must be present in the "Authorization" header of the request, and any additional data (such as the new user's username and password) must be present in the JSON object of the request.

The API application is in a Python script, created as a CLI, which takes arguments for connection parameters to the database.

4 Usage

Docker [3] is being used for modularization of the database and the API, where both are organized into sub-folders. Each sub-folder contains a *Dockerfile*, which can be executed by Docker to create a container image. Each container can be executed independently, giving a higher modularity to the application. All the commands for building and executing the containers are in the *Makefile* file, which can be executed by the *make* command in Unix.

5 References

References

[1] LLC. Assurance Technologies. *Passlib 1.7.1 documentation — Passlib v1.7.1 Documentation.* URL: `https://passlib.readthedocs.io/en/stable/`.

[2] Kevin Burke et al. *Flask-RESTful — Flask-RESTful 0.3.7 documentation.* URL: `https://flask-restful.readthedocs.io/en/latest/`.

[3] *Empowering App Development for Developers | Docker.* URL: `https://www.docker.com/`.

[4] Miguel Grinberg. *Welcome to Flask-HTTPAuth's documentation! — Flask-HTTPAuth documentation.* URL: `https://flask-httpauth.readthedocs.io/en/latest/`.

[5] The PostgreSQL Global Development Group. *PostgreSQL: Documentation: 11: 12.1. Introduction.* URL: `https://www.postgresql.org/docs/11/textsearch-intro.html`.

[6] The PostgreSQL Global Development Group. *PostgreSQL: Documentation: 11: 66.1. Introduction.* URL: `https://www.postgresql.org/docs/11/gin-intro.html`.

[7] The PostgreSQL Global Development Group. *PostgreSQL: Documentation: 11: 9.13. Text Search Functions and Operators.* URL: `https://www.postgresql.org/docs/11/functions-textsearch.html`.

[8] Pallets. *Welcome to Flask — Flask Documentation (1.1.x).* URL: `https://flask.palletsprojects.com/en/1.1.x/`.

[9] Pallets. *Werkzeug | The Pallets Projects.* URL: `https://palletsprojects.com/p/werkzeug/`.

[10] NLTK Project. *Natural Language Toolkit - NLTK 3.4.5 documentation.* URL: `https://www.nltk.org/`.

[11] André Moreira Souza. *andremsouza/docx2json: Python algorithm that converts raw text from a .docx file into .json format.* URL: `https://github.com/andremsouza/docx2json`.

[12] Daniele Varrazzo. *PostgreSQL + Python | Psycopg.* URL: `http://initd.org/psycopg/`.