Modelado de familias y derecho a la ciudadanía Italiana en bases de datos de grafos

Agustín Bologna
Facultad de Ingeniería, Universidad de la República
Montevideo, Uruguay
agustin.bologna@fing.edu.uy

Andres Collares
Facultad de Ingeniería, Universidad de la República
Montevideo, Uruguay
andres.collares@fing.edu.uy

***Resumen*—El proyecto consiste en generar una base de datos de grafos de familias, con el objetivo de clasificar a las personas en base a su posibilidad de obtener la ciudadanía Italiana. Una vez cargados los datos se implementa un sitio web que los consume de forma de clasificar a cada persona y presentar el resultado mediante una interfaz gráfica. Por último, se realizan pruebas de carga sobre la API generada.**

I. Introducción

En Uruguay es frecuente el interés de la población por obtener la nacionalidad de algún país Europeo, la mayoría de las veces estos trámites son muy engorrosos, largos y difíciles de realizar, el objetivo del presente proyecto es diseñar una base de datos de grafos que modele árboles familiares, e implementar una aplicación web de forma de poder clasificar a cada integrante de una familia basándonos en si tienen derecho a la ciudadanía Italiana o no, y porque vía (administrativa o judicial).

Para poder determinar si una persona tiene derecho a la ciudadanía se deben cumplir un conjunto de requisitos:

- Que exista un ancestro Italiano que estuvo vivo posterior al 1861.
- Que no haya renunciado a su ciudadanía Italiana (tomando otra), aunque en caso de que haya renunciado luego del nacimiento del hijo, el hijo nace con el derecho a la ciudadanía.
- Si cualquier ancestro (línea directa) nace en otro país, se necesita saber que no se nacionalizó en ese país (no es necesario para hijos nacidos después de 1992).
- La ciudadanía Italiana era heredable solo por padre y no por madre hasta 1948, cuando surge una ley por la cual las mujeres obtienen el derecho de heredarle a sus hijos e hijas el derecho a la ciudadanía Italiana. Esto determina la vía por la cual se puede obtener la ciudadanía, dado que en caso de que el hijo no haya heredado el derecho por parte de madre, se puede realizar un juicio en Italia mediante el cual toda la familia obtiene la ciudadanía. En otro caso se puede realizar un trámite administrativo (sin necesidad de juicio) en la embajada de Italia.

Para el desarrollo de la aplicación se utilizará **Neo4j** como base de datos de grafos, **Django** como *web framework* y **Locust** (IV-A) para realizar las pruebas de carga con el objetivo de determinar la forma en la que se comporta la aplicación en distintos niveles de concurrencia en función del tiempo de respuesta.

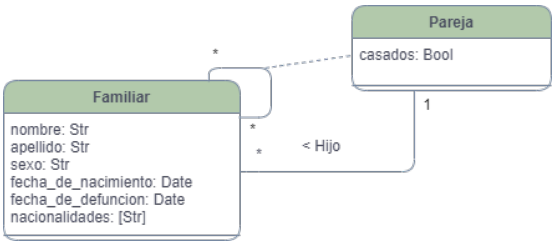


Figura 1: Diagrama parcial del diseño relacional.

II. Trabajos relacionados

En el mercado existen múltiples productos que giran en torno a las familias y genealogía, como FamilySearch[1], FindmyPast[2] y Ancestry[3], en donde trabajan al rededor del mundo equipos de investigadores con el objetivo de generar bases de datos valiosas para familias, historiadores, empresas relacionadas a la genealogía y demás.

El principal valor de estos productos es la información, dado que no ofrecen servicios de procesamiento de datos ni permiten ejecutar consultas complejas sobre los mismos.

III. Desarrollo del proyecto

III-A. Obtención y anonimización de los datos

Dada la dificultad de generar árboles familiares aleatorios que tengan una estructura coherente, para las pruebas de carga se utilizarán datos provenientes de una base de datos relacional perteneciente a la empresa Gestari SAS. Esta base de datos contiene información de 1087 familias y 20330 personas, donde la estructura que modela a los familiares toma la forma que se aprecia en la figura 1.

Para cargar esta información a la base de datos de Neo4j fue necesario crear un script en Python que genera un archivo JSON con los datos necesarios, con la estructura recursiva presente en el Listado 1.

Para anonimizar los datos se quitan todos aquellos que identifican a una persona y se utiliza la biblioteca *names* [1]

[1]https://www.familysearch.org/en/
[2]https://www.findmypast.com/
[3]https://www.ancestry.com/

de Python para generar nombres aleatorios, la misma provee el método get_full_name que recibe como parámetro el género de la persona y devuelve un nombre aleatorio.

Listado 1 Formato de los datos extraídos

```
[ [ {
   "sex": "MALE",
   "birthday": "1820-01-01",
   "date_of_death": null,
   "has_citizenship": true,
   "citizenship_resignation_date": null,
   "family_uuid": "{uuid4}",
   "partners": [
      {
         "sex": "FEMALE",
         "birthday": "1823-12-10",
         ...,
         "family_uuid": "{uuid4}",
         "is_married": false,
         "offspring": [
            {
               "sex": "MALE",
               ...,
               "partners": [...]
            }, ...]
      }, ...]
   }, ...]
]
```

III-B. Diseño y carga de datos

Se discutieron múltiples diseños y se optó por el mejor para recorrer el árbol transversalmente desde arriba hacia abajo, dado que es el camino que se debe recorrer para vincular a un antepasado con su descendiente. En el mismo se cuenta con nodos *Person* para representar a las personas de cada familia, el cuadro I muestra en detalle los atributos modelados, para representar una pareja se crea la relación *PARTNER* que va de *Person* a *Person* conteniendo un atributo de tipo *bool* para indicar si las personas están casadas. Por último, para modelar la descendencia de cada persona se crea la relación *OFFSPRING*, que vincula padres con hijos (Figura 2).

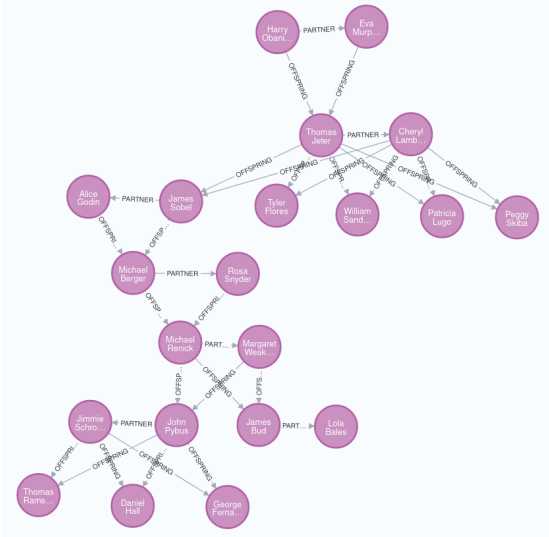


Figura 2: Representación de un árbol familiar.

Para definir esta estructura se utiliza la biblioteca **Neomodel**[4], la misma provee las clases *DjangoNode*, *StructuredNode* y *StructuredRel* para generar interfaces de creación, actualización, eliminación y listados de nodos y relaciones con la base Neo4J.

Una vez instanciadas las clases de *Django* se creó un script para la importación de los datos a partir del archivo *families.json* (Listado 1), el cual genera todos los nodos y relaciones utilizando las interfaces de *Neomodel* [2].

Listado 2 ejemplos de consultas utilizando neomodel

```
# creacion de un nodo
person_instance = person(
   name="persona anonima",
   family_uuid=uuid4,
   sex="female",
   has_citizenship=true,
)

# creacion de una relacion entre nodos
person_instance.partner.connect(
   partner_instance,
   {"is_married": is_married}
)

# como obtener una familia
person.nodes.filter(family_uuid="{family_uuid}")

# misma consulta utilizando la interfaz con cypher
from neomodel.match import db
query = "match(p:person) return distinct p.family_uuid"
results, meta = db.cypher_query(query)
family_uuids = [row[0] for row in results]
```

III-C. Desarrollo de la API y aplicación web

Se desarrollaron cuatro *endpoints*.

III-C1. Familias en el sistema: Retorna una lista con todos los id's de familia únicos en el sistema y se encuentra en la URL base.

III-C2. Ingresar familia en el sistema: Realizando una solicitud de tipo POST con cuerpo *form-data* de tres listas: 'members', 'relations_partners', 'relations_offsprings', donde la primera contiene los nodos a crear, la segunda una lista de pares de ids representando las relaciones de pareja y la tercera una lista de pares de ids representando las relaciones de descendientes, se crean todos los nodos y relaciones y se retorna el *uuid* correspondiente a la familia creada. Se encuentra en */create_family/*

III-C3. Vista de familia beta y final: Ambas versiones proveen una interfaz amigable para visualizar un árbol familiar (Figura 3), donde el borde verde indica que una persona es ciudadano, el borde amarillo que tiene derecho a la ciudadanía por vía administrativa, púrpura si es por vía judicial y rojo en caso de que no tenga derecho. La versión *beta* recorre dos veces el árbol, durante la primera lo clasifica y en la segunda genera el JSON que consume la biblioteca *dTree*[5] para dibujar el árbol, mientras que la versión final realiza ambos cálculos al mismo tiempo. Se encuentran en */process_family_beta/{family_uuid}/* y */process_family/{family_uuid}/* respectivamente.

[4]https://neomodel.readthedocs.io/en/latest/
[5]https://github.com/ErikGartner/dTree

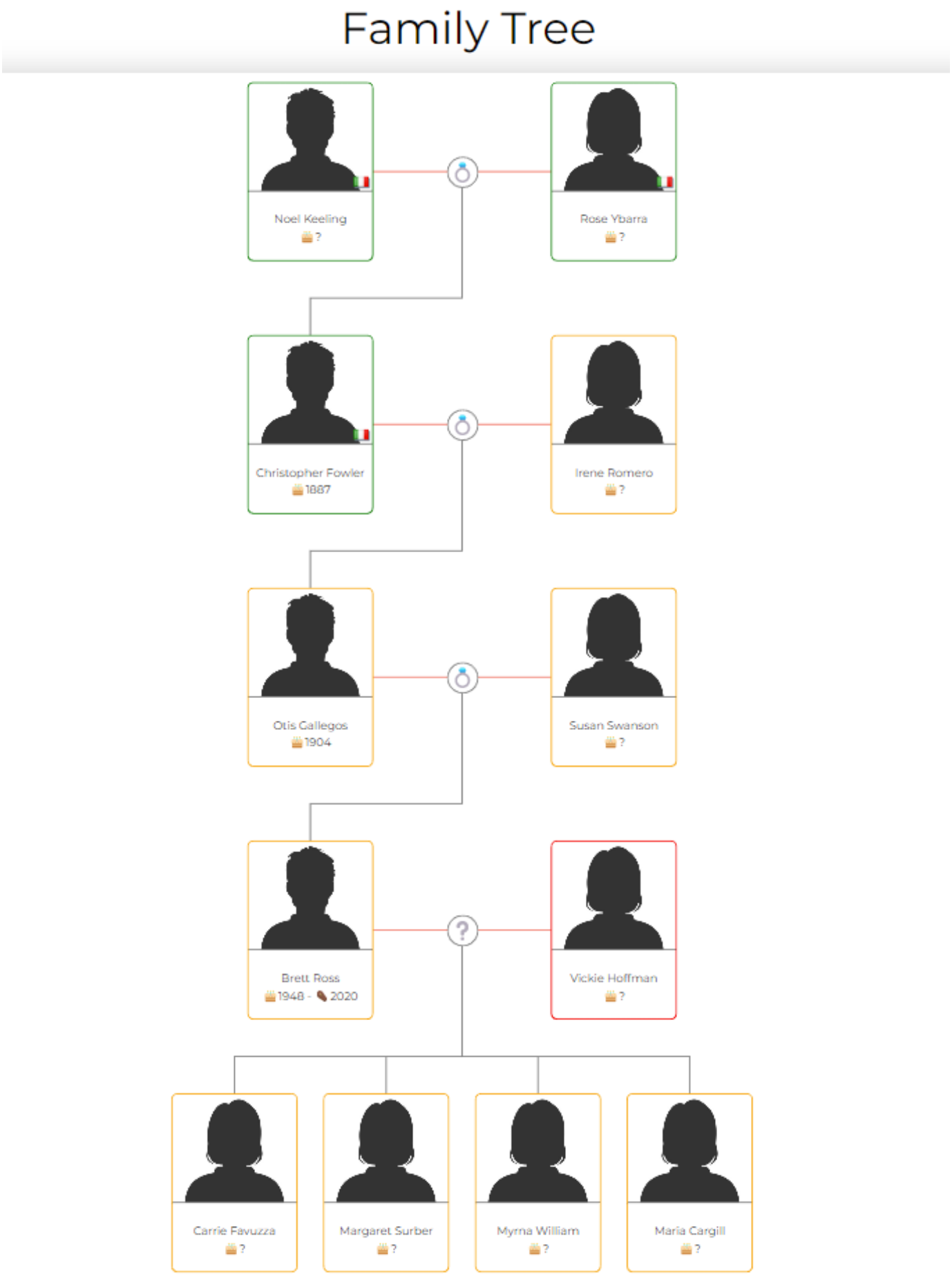


Figura 3: Vista de familia, árbol 66f40507-0c78-41e7-864a-442d1c50e537.

Cuadro I: Atributos de un nodo Person

Atributo	**Tipo**	**Requerido**	**Descripción**
name	*String*	✓	Nombre completo
family_uuid	*String*	✓	Identificador de familia
sex	{*'MALE', 'FEMALE', 'OTHER'*}	✓	Sexo
birthday	*Date*		Fecha de nacimiento
date_of_death	*Date*		Fecha de defunción
has_citizenship	*Boolean*	✓	¿Tiene ciudadanía?
citizenship_resignation_date	*Date*		Fecha de renuncia a la ciudadanía

Listado 3 Función de clasificación de un árbol familiar (versión beta)

```
def possible_citizenship(person, could_get_citizenship):
   if person.has_citizenship or could_get_citizenship.get(person.id) or person.citizenship_resignation_date:
      # Se verifican las condiciones de heredar la ciudadania
      if person.date_of_death and person.date_of_death < date(1861, 1, 1):
         return could_get_citizenship
      if could_get_citizenship.get(person.id):
         administrative = could_get_citizenship.get(person.id)["administrative"]
      elif person.sex == "FEMALE" and person.date_of_death and person.date_of_death < date(1948, 1, 1):
         administrative = False
      else:
         administrative = True

      # Si se cumplen las condiciones entonces se verifica la ciudadania de los hijos
      offspring = person.offspring.all()
      for member in offspring:
         if person.citizenship_resignation_date == None or (
            person.citizenship_resignation_date and member.birthday < person.citizenship_resignation_date
         ):
            if not could_get_citizenship.get(member.id):
               could_get_citizenship[member.id] = {
                  "member": member.id,
                  "administrative": administrative,
               }
            could_get_citizenship = possible_citizenship(member, could_get_citizenship)
   return could_get_citizenship
```

Listado 4 Función de clasificación de un nodo (versión final)

```
def value_citizenship(person, parent_rel, citizenship_state):
   if citizenship_state == "no":
      for parent in [parent_rel.start_node(), parent_rel.end_node()]:
         if (parent.citizenship_resignation_date and
            (person.birthday < parent.citizenship_resignation_date or person.birthay > date(1992, 1, 1)))
            or parent.has_citizenship and not (person.date_of_death and person.date_of_death < date(1861, 1, 1)
            or person.citizenship_resignation_date)):
            # caso 3
            return "admin"
      return "no"
   else:
      if person.has_citizenship:
         if person.date_of_death and person.date_of_death < date(1861, 1, 1) or person.citizenship_resignation_date:
            # caso 1 y 2
            return "no"
         elif person.sex == "female" and person.date_of_death and person.date_of_death < date(1948, 1, 1):
            # caso 6
            return "trial"
         else:
            return "admin"
      elif person.citizenship_resignation_date:
         # caso 5
         return "no"
      else:
         # se continua con el mismo estado
         return citizenship_state
```

III-C4. Administrador: Además de las vistas anteriores, se configuró un administrador que permite visualizar el listado de *Person* ordenado por familia y fecha de nacimiento. El mismo está disponible en */admin/*.

III-D. Algoritmo de clasificación

Para clasificar un árbol se lo recorre partiendo desde los nodos que no tienen relaciones entrantes, luego moviéndose hacia sus parejas y desde cada pareja hacia sus hijos, continuando de esta forma hasta llegar a los nodos que no tuvieron hijos.

En la versión beta existía una función encargada de clasificar el árbol completo, la misma se encuentra en el listado 3.

Para evaluar a una persona mientras se procesan los datos del árbol es necesario tener sus datos, tener los datos de sus padres y cuál es el estado actual del derecho a la ciudadanía de su rama en el árbol, con esta información es posible evaluar los requisitos para la ciudadanía y definir si corresponde o no, la función de valuación se encuentra en el listado 4.

Aunque la función beta es más legible que la función de clasificación final, se esperan mejores resultados con esta última, dado que se realizan aproximadamente la mitad de consultas que en la versión beta.

IV. Pruebas de carga

IV-A. Locust

Locust[6] es una herramienta que permite crear pruebas de carga con facilidad, es de las más utilizadas para realizar esta tarea (19.2k *stars* en github) y las pruebas son configuradas en python.

IV-B. AuraDB

En un principio se utilizó la versión gratuita de AuraDB para manejar la base de datos, la primera prueba realizada fue con un único usuario, donde el mismo creaba una familia aleatoria 50 veces y el promedio de respuesta fue de 12.16 segundos (cuadro II).

Dado que el resultado fue muy negativo, se realizó la misma prueba, pero esta vez en un ambiente local, como resultado el promedio de respuesta paso a 164ms (cuadro III), por lo que de este punto en adelante el trabajo se realizó solo en el ambiente local.

IV-C. Diseño de las pruebas de carga

Para ejecutar la aplicación se utiliza *gunicorn*[7], de forma de poder atender solicitudes utilizando múltiples hilos.

Como base para las pruebas se elige un segundo como el máximo tiempo de respuesta aceptable [3] y 261ms o menor como un tiempo de respuesta excelente [4].

Características del ambiente de prueba local:

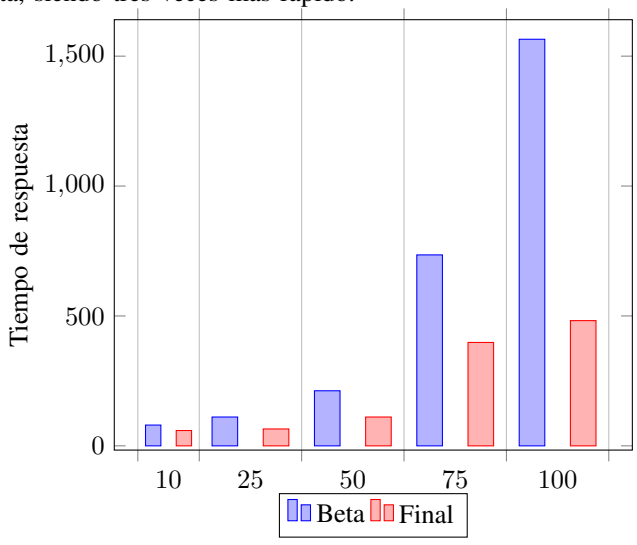
- OS: Ubuntu 22.04 desde WSL en Windows 11 21H2
- CPU: Intel core i5 9600k (6 Core & 6 Threads @ 4.6GHz)
- RAM: 64Gb 3200MHz
- SSD: Samsung 970 EVO Plus

Se diseñaron las siguientes pruebas de carga para determinar la cantidad de usuarios concurrentes que soporta la aplicación.

Se tienen 2 tipos de usuarios, el primero *(CreateUser)*, que se encarga de crear registros nuevos en la base de datos cada 120 a 180 segundos y el segundo *(QueryUser)*, que se encarga de realizar consultas sobre las familias del sistema cada 1 a 3 segundos. Para determinar el número de usuarios concurrentes que pueden utilizar la aplicación se realizaran varias pruebas de carga, cada una con una cantidad fija de usuarios, con una distribución de 20 % de *CreateUser* y 80 % de *QueryUser* y una duración de diez minutos.

Como se puede observar en el cuadro IV, utilizando la versión beta de la consulta, el sistema soporta hasta cincuenta usuarios concurrentes sin problema alguno. Al aumentar la cantidad de usuarios concurrentes a setenta y cinco se comienza a notar demoras que están cerca de no ser aceptables para los estándares de hoy en día. Por último se aumentó la cantidad de usuarios a cien, en este punto las demoras son importantes, dado que el tiempo de respuesta promedio es 1.56 segundos y el límite de tiempo de respuesta máximo aceptable lo tomamos como un segundo, la versión beta no es aceptable para un sistema con cien usuarios concurrentes.

Al ejecutar las pruebas sobre la versión final se puede apreciar una baja drástica en el tiempo de respuesta frente al algoritmo de la versión beta, el punto de más contraste es el caso de los cien usuarios concurrentes, donde el algoritmo final es capaz de resolver las consultas en un tiempo promedio casi excelente, de 482ms frente a los 1565ms de la versión beta, siendo tres veces más rápido.



V. Conclusiones y trabajo futuro

V-A. Conclusiones

Neo4j probó ser una excelente solución como base de datos para el caso de los árboles familiares, obteniendo un excelente resultado con hasta cien usuarios concurrentes.

El desarrollo de la aplicación utilizando *Django* y *Neomodels* fue muy intuitivo y moderno debido a las herramientas

[6]https://github.com/locustio/locust
[7]https://gunicorn.org/

Cuadro II: único usuario, creación de una familia aleatoria con optimizaciones

Requests	Median (ms)	90 %ile (ms)	99 %ile(ms)	Average (ms)	Min (ms)	Max (ms)
50	12000	12000	14000	12160	11984	14136

Cuadro III: único usuario, creación de una familia aleatoria en ambiente local

Requests	Median (ms)	90 %ile (ms)	99 %ile(ms)	Average (ms)	Min (ms)	Max (ms)
50	140	200	600	164	121	600

Cuadro IV: Pruebas de carga con duración de 10 minutos (beta)

CreateUser	QueryUsers	Requests	Median(ms)	90 %ile(ms)	99 %ile(ms)	Average(ms)	Min(ms)	Max(ms)
2	8	1522	51	170	440	80	5	789
5	20	3765	75	240	640	111	5	1244
10	40	7323	130	540	1000	212	5	1525
15	60	9152	580	1600	2600	735	5	3440
20	80	9758	1500	2600	3300	1565	7	3846

Cuadro V: Pruebas de carga con duración de 10 minutos (final)

CreateUser	QueryUsers	Requests	Median(ms)	90 %ile(ms)	99 %ile(ms)	Average(ms)	Min(ms)	Max(ms)
2	8	1561	40	120	390	59	3	571
5	20	3847	46	130	370	65	3	661
10	40	7422	67	260	640	111	3	1202
15	60	9983	230	1100	2000	398	3	2733
20	80	12738	430	970	1500	482	3	1789

que estas presentan, lo cual demuestra lo baja que es la barrera para migrar a una base de datos de grafos como lo es *Neo4j*.

V-B. Trabajo Futuro

Realizar pruebas con una mayor cantidad de nodos y relaciones utilizando una licencia *pro* de AuraDB, para verificar si la arquitectura planteada es escalable.

Mejorar la experiencia de usuario de la aplicación, como por ejemplo, tener una forma interactiva de ingresar los miembros de la familia, crear un sistema de autenticación.

Todos los programas implementados, instrucciones de uso y datos se encuentran en el repositorio https://github.com/Etzior/nsql_grupo23.

REFERENCIAS

[1] "Names." https://pypi.org/project/names/.

[2] "Advanced queries¶." https://neomodel.readthedocs.io/en/latest/queries.html#complex-lookups-with-q-objects.

[3] W. L. in Research-Based User Experience, "Response time limits: Article by jakob nielsen." https://www.nngroup.com/articles/response-times-3-important-limits/.

[4] "What is the average server response time?." https://www.littledata.io/average/server-response-time.