Universidad de la República

Facultad de Ingeniería

Integración de datos desde BPMS y fuentes NoSQL para minería de procesos de negocio

Informe de Proyecto de Grado presentado por

Santiago Sosa, Germán González

como requisito de graduación de la carrera de Ingeniería en Computación de Facultad de Ingeniería de la Universidad de la República

Supervisores

Dra. Ing. Andrea Delgado
Dr. Ing. Daniel Calegari

Montevideo, 22 de diciembre de 2022

Resumen

En entornos organizacionales de gran escala como por ejemplo e-government, las organizaciones enfrentan diversos desafíos para gestionar grandes volúmenes de datos, siendo uno de los más importantes obtener información y valor desde los datos en sus sistemas de información. Los procesos de negocio que realizan las organizaciones pueden ser implementados y ejecutados en plataformas BPM (Business Process Management System, BPMS) que proveen soporte al modelado, desarrollo, operativa y mejora de dichos procesos. Tanto las organizaciones, como sus procesos y los sistemas de software que soportan dichos procesos y datos, son cada vez más complejos, definiendo ecosistemas en los que se hace necesario integrar diferentes visiones, técnicas y herramientas para la gestión de la información, procesos y sistemas asociados.
La integración de datos incluye diversas fuentes, bases de datos relacionales y no relacionales (NoSQL), asociadas a los distintos sistemas en la organización, incluyendo tanto procesos que ejecutan en plataformas BPMS, como sistemas de información tradicionales, redes sociales, servicios internos y externos, entre otros. La Minería de Procesos de Negocio (*Process Mining*) se basa en técnicas de minería de datos para analizar los registros de eventos (*logs*) asociados a la ejecución de los procesos en las organizaciones; por ejemplo mediante la extracción de logs de auditorías en la base de datos, o en base a trazas de ejecución en sistemas o plataformas de ejecución de procesos de negocio (BPMS).

En este proyecto se propone integrar y extender trabajos anteriores en integración de datos de procesos y organizacionales desde fuentes de datos NoSQL y plataformas BPMS de soporte a sistemas de información basados en procesos. El objetivo principal del proyecto es automatizar la extracción de datos desde fuentes NoSQL hacia un metamodelo definido previamente que provee la base para integrarlos con los datos de procesos.

Para ello, se define una API genérica que permite la extracción de datos desde diversas fuentes NoSQL. Además, la API se implementa para MongoDB y Neo4j y se pone en practica en un caso de estudio de la literatura.

Palabras clave: NoSQL, BPMS, minería datos, MongoDB, Neo4j

Índice general

Capítulo 1

Introducción

La gestión de procesos de negocio (*Business Process Management*, BPM) [1] es un tema que ha tenido un incremento importante debido a que es muy aplicable desde un punto de vista práctico y, al mismo tiempo, ofrece muchos retos para los desarrolladores de software y *Data Scientist*[2]. Tradicionalmente, los sistemas de información utilizaban el modelado de la información como punto de partida, es decir, los enfoques basados en datos han dominado el panorama de los sistemas de información. Sin embargo, en el último tiempo ha quedado claro que los procesos son igualmente importantes y deben ser apoyados de forma constante. Los sistemas de gestión de *workflow*, con un auge en la década del 90, tenían como objetivo la automatización de procesos estructurados. Por ello, su uso se limitó a pocos ámbitos. Sin embargo, los conceptos básicos del flujo de trabajo han sido adoptados por diferentes tipos de sistemas de información. BPM aborda el tema del apoyo a los procesos desde una perspectiva más amplia, incorporando diferentes tipos de análisis (por ejemplo, simulación, verificación y minería de procesos) y vinculando los procesos a los aspectos organizacionales y sociales. Además, el interés actual por el BPM se ve alimentado por los avances tecnológicos (arquitecturas orientadas a servicios) que desencadenan los esfuerzos de normalización, derivando en la creación de lenguajes estándar como es el ejemplo de *Business Process Model and Notation* (BPMN) [3].

Por otro lado, el uso masivo de aplicaciones genera grandes volúmenes de información que tienen una diversa cantidad de usos al aplicar técnicas de Minería de Procesos [4]. Los datos deben ser almacenados en estructuras acordes que soporten esta magnitud de datos. Las bases de datos NoSQL son almacenamientos de información que no cumplen las restricciones básicas del clásico modelo relacional [5] y que por tanto, no tienen un conjunto de tablas relacionadas entre sí de una manera ordenada y lógica. Las bases de datos NoSQL surgen por las deficiencias encontradas en estos modelos relacionales, presentando dificultades para manejar grandes cantidades de información de una manera rápida y eficaz. En la actualidad, los volúmenes de información crecen a un ritmo sin precedentes, y cada vez se hace más compleja su administración. En este escenario, las organizaciones no solo desean almacenar esta información, sino que quieren

sacarle el mayor provecho. Alineados con estos cambios, los usuarios piden cada vez más velocidad en las consultas, y la arquitectura de los sistemas ha tenido un cambio considerable para hacer frente a estos requerimientos.

El presente trabajo apunta a encontrar un mecanismo que permita descubrir la relación existente entre datos surgidos de los procesos de negocio y datos puramente organizacionales, siguiendo la línea de investigaciones anteriores [6]. A este trabajo se le agrega una dificultad extra que es que las bases de datos organizacionales en la actualidad no solo se limitan a almacenar los datos en bases de datos relacionales, sino que la tendencia los lleva a que las organizaciones adopten cada vez más la alternativa del almacenamiento en bases de datos NoSQL. Es aquí que surge el interés de investigar esta forma de almacenar los datos e incorporarlos a la línea de trabajo. En este documento abordamos la integración de los datos de los procesos de negocio y de la organización utilizando bases de datos NoSQL con el objetivo de brindar a las organizaciones elementos que permitan una evaluación completa de la ejecución de los procesos de negocio. Proporcionamos un análisis basado en métricas que ayuden a dar foco en los tipos de bases de datos NoSQL más populares y sus manejadores asociados [7]. Además, se busca adaptar y extender implementaciones de proyectos anteriores [8] que permitan cumplir un ciclo completo de extracción de datos de diferentes fuentes de una manera genérica.

A continuación se presentan los objetivos principales:

- Definir un mecanismo de representación común de la información contenida en bases de datos NoSQL.
- Definir una API genérica que permita extraer información desde diversas bases de datos NoSQL.
- Implementar la API con tecnologías existentes e integrarla al marco general de trabajo, incluyendo las adaptaciones que sean necesarias.

Dentro de los desafíos más importantes que presenta el proyecto se destaca la necesidad de desarrollar un proceso genérico que permita tener como punto de partida una base de datos NoSQL y poder mapear sus datos a un esquema general representado en una base de datos relacional. Este esquema general viene dado por trabajos relacionados donde se centran en la generación de modelos y algoritmos para la minería de procesos y datos [9].

El informe está organizado en seis capítulos principales. En el capítulo 2 se presentan los conceptos teóricos necesarios para comprender las siguientes secciones. Estos son Procesos de negocio, lenguaje BPMN y bases de datos NoSQL. En este capítulo también se analizan las distintas bases de datos y se seleccionan los manejadores a utilizar en ejemplos de secciones posteriores.

En el capítulo 3 se analiza el problema general a resolver, detallando cómo se abordará en el proyecto para cumplir con los objetivos planteados. Este capítulo se divide en tres secciones, cada una de ellas asociada a los objetivos generales del proyecto.

En el capítulo 4 se propone la solución al problema definido en el capítulo anterior, detallando cada uno de los componentes utilizados para cumplir con

los objetivos. Nuevamente como en el caso anterior, el capítulo se divide tres en secciones, siendo cada una de éstas la propuesta de solución para los objetivos generales.

En el capítulo 5 se muestran los resultados obtenidos luego de utilizar el ejemplo de *Student Mobility*[6].

Finalmente en el capítulo 6 se presentan las conclusiones del proyecto y se mencionan trabajos futuros que se pueden desarrollar a partir de la solución alcanzada.

Capítulo 2

Marco teórico

Este capítulo da un marco teórico general sobre los distintos conceptos previos requeridos para el proyecto, comenzando desde la definición y selección del motor de procesos hasta la investigación de las diferentes variedades de bases de datos no relacionales. El resultado de este marco teórico es determinar cuales son las bases de datos mas relevantes para darle soporte en el proyecto. Para esto último se recurrió a un sitio web donde se hace un ranking bases de datos de distintos tipos y a la consultora Forrester [10].

2.1. Procesos de negocio

Un Proceso de Negocio (BP, del inglés Business Process)[11] es un conjunto de actividades que de manera coordinada buscan alcanzar un objetivo de negocio. La variabilidad en un BP es la posibilidad de elegir entre diferentes opciones al momento de tomar una decisión en el diseño o particularización de un proceso. Entre los lenguajes utilizados para el modelado de un BP se encuentra *Business Process Model and Notation 2.0* (BPMN 2.0)[12]. Es el lenguaje estándar definido por la *Object Management Group* (OMG)[13]. BPMN, y en particular su versión 2.0 proporciona a las organizaciones la capacidad de entender los procedimientos de negocio internos utilizando una notación gráfica y brindando la capacidad de comunicar estos procedimientos de una manera estándar. El lenguaje provee un amplio conjunto de elementos para representar el común de las situaciones, facilitando la comunicación entre involucrados, y promoviendo la interoperabilidad con formato estándar XML.

A continuación se plantea un ejemplo práctico. El objetivo del mismo es aplicar las definiciones anteriores utilizando la notación BPMN 2.0. El ejemplo trata de un proceso de compra de un producto en un comercio electrónico, y puede ser reducido en los siguientes seis pasos:

1. El cliente realiza un pedido de un producto en línea

2. La empresa registra el pedido y pago

3. La empresa produce el producto o la retira del stock
4. Empaca el producto
5. La envía al transporte de mercancías para la entrega de pedidos
6. Finaliza el pedido

A través de la notación BPMN, el proceso se puede representar como se muestra en la figura 2.1

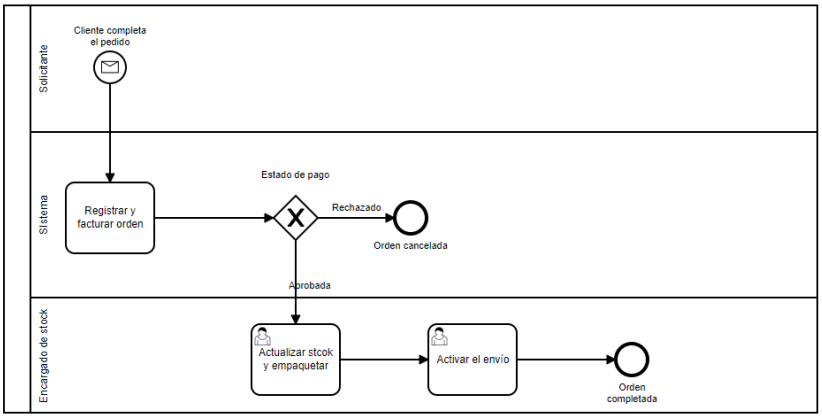

Figura 2.1: Proceso de ejemplo

Si bien es un ejemplo simplificado, alcanza para mostrar que fragmentar los procesos de una misma tarea puede facilitar su comprensión y la identificación de sus excepciones o caminos alternativos.

Dentro del diagrama de procesos mapeados, existen cinco categorías que representan flujos internos de cada proceso. Cada categoría tiene elementos con iconos que simbolizan las tareas a realizar, conexiones entre procesos, desvíos, eventos, entre otros. Como complemento a la notación BPMN 2.0 presentada se agregan las definiciones de *swimlanes*, *pools* y *lanes*. Estos conceptos serán de utilidad para la aplicación de la solución del capítulo 5. El término *swimlanes* en BPMS sirve para ayudar a dividir y organizar actividades en un diagrama. De éstos hay dos tipos principales:

- *Pools* - Actúan como contenedores para un proceso, cada uno representando un participante en un diagrama de procesos de negocio colaborativo. Representan participantes.
- *Lanes* - Utilizados a menudo para representar roles de negocio internos dentro de un proceso, los lanes en realidad proveen un mecanismo genérico para particionar los objetos dentro de un *pool*, basados en las características del proceso o elementos.

El término *pool* se obtuvo ampliando la analogía de *swimlane*. Una piscina de natación tiene carriles. BPMN tiene dos tipos de particiones *swimlane* y un tipo está incluido en el otro. Por lo tanto, se llaman *lanes* a las sub particiones y, a la partición que contiene los *lanes* se le llama *pool*.

En BPMN, los *pools* representan participantes en un diagrama de procesos de negocio iterativo o colaborativo. Un participante se define como un rol de negocios en general. Los *pools* son representados como una caja rectangular que actúa como contenedor para los objetos de flujo. En el caso del ejemplo se representa un *pool* con los *lanes* Solicitante, Sistema y encargado de Stock.

En [14] se presenta un póster informativo de resumen donde contiene los elementos más importantes de la notación.

2.1.1. BPMS

Los BPMS (*Business Process Management Systems*, por sus siglas en inglés) son sistemas para gestionar BP, desde su creación, puesta en funcionamiento, seguimiento y modificación; en definitiva permiten controlar el ciclo de vida completo de estos.

En la industria existe una gran cantidad de estos sistemas que varían en cuanto a funcionalidades y licenciamiento. En proyectos anteriores se utilizaron JBPM[15], Bonita[16], Camunda[17] y Activiti [18]. Entre estas opciones elegimos Camunda, ya que posee una API muy completa con documentación detallada [19], siendo uno de los factores más importantes a la hora de elegir dicho motor para el proyecto.

2.1.2. Camunda

Es una plataforma de BPM que surge como un *fork* de Activiti [18]. El objetivo de esta plataforma es promover la innovación en la automatización de procesos con un enfoque basado en estándares, altamente escalable y colaborativo para el negocio. Ofrece una versión *Open Source*, así como también una edición *enterprise*. En este proyecto se utiliza la versión gratuita más reciente al momento de iniciar, la 7.16. Además de la aplicación web, provee una API REST para iniciar procesos, asignar y completar tareas, entre otros. En la figura 2.2 se presenta un esquema de la arquitectura típica de Camunda.

2.2. Bases de datos NoSQL

Hasta hace poco tiempo, los sistemas de administración y gestión de bases de datos relacionales (RDBMS) eran utilizados para administrar todo tipo de datos, independientemente de su ajuste natural al modelo de datos relacional. La aparición de la web 2.0, la nube, *Big Data* y los dispositivos móviles generaron la necesidad de un nuevo tipo de base de datos que permitiera un mejor rendimiento al tratar grandes cantidades de escritura de datos o donde el particionamiento tuviera una importancia central. Esto se debe a que las bases de

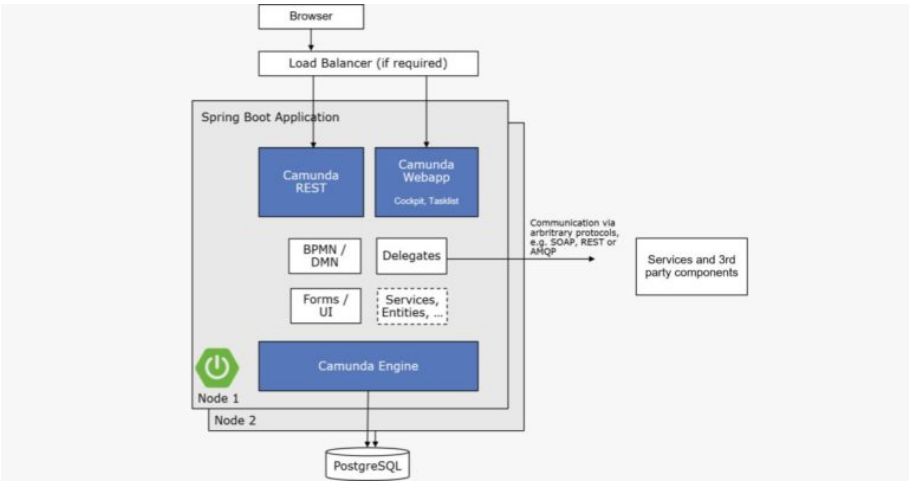

Figura 2.2: Arquitectura típica de Camunda [20]

datos relacionales no cumplían con dichas características debido a la rigidez de representación.

Aplicaciones como el análisis de archivos de log en tiempo real, el comercio electrónico, los servicios basados en la ubicación, entre otras, exhiben una preponderancia de operaciones de inserción y recuperación a gran escala, requiriendo alta concurrencia y baja latencia en las escrituras. Se encontró que las bases de datos relacionales eran inadecuadas para manejar la escala y la estructura variada de los datos en estos tipos de aplicaciones, lo que marcó el comienzo de una serie de opciones para la gestión de *Big Data* bajo el término general NoSQL.

NoSQL describe una gran clase de bases de datos que no tienen las propiedades de las bases de datos relacionales tradicionales y, por lo general, no se consultan utilizando el lenguaje SQL. Los sistemas NoSQL proporcionan partición y replicación de datos como características integradas, apuntando a tener alta escalabilidad y disponibilidad. Un sistema distribuido, dentro de las prestaciones principales, puede ofrecer consistencia, disponibilidad y tolerancia al particionamiento. El teorema CAP indica que un sistema distribuido solamente puede garantizar como máximo dos de las tres características [21]. Es por esto que los sistemas NoSQL suelen relajar las propiedades ACID, en particular la consistencia, priorizando la disponibilidad.

El desarrollo de aplicaciones que utilizan sistemas NoSQL es bastante diferente del proceso que se utiliza con RDBMS. Los enfoques de bases de datos NoSQL comienzan identificando los tipos de consultas de las aplicaciones y estructurando el modelo de datos para respaldar estas consultas de manera eficiente. En otras palabras, existe un fuerte acoplamiento entre el modelo de datos y las consultas de la aplicación. Cualquier cambio en las consultas requerirá cambios en el modelo de datos. Este enfoque está en marcado contraste con

los principios de RDBMS basados en la independencia de datos físicos y lógicos. [23]

2.2.1. Características

Gran parte de las aplicaciones que dieron inicio a las bases de datos NoSQL se caracterizan por alojar datos masivos que deben fusionarse desde múltiples fuentes y procesarse en tiempo real. A continuación se describen algunas de las características más importantes de este tipo de bases de datos.

Big Data

Los datos que son demasiado grandes y complejos para capturar, almacenar, procesar, analizar e interpretar utilizando las herramientas y métodos de última generación se denominan *Big Data*. Se caracteriza por "las cinco Vs": volumen, velocidad, variedad, valor y veracidad. El tamaño de los datos (volumen) es del orden de terabytes o incluso petabytes, y se dirige rápidamente hacia los exabytes. La velocidad se refiere a la velocidad a la que se generan los datos. El *Big Data* empresarial suele ser heterogéneo (variedad) y se compone de datos estructurados, semi estructurados y no estructurados. El valor se refiere a la extracción de información útil y procesable de conjuntos de datos masivos. Debido a la diversidad en las fuentes y la evolución de los datos a través del procesamiento intermedio, los problemas de seguridad, privacidad, confianza y responsabilidad deben abordarse utilizando la procedencia segura de los datos (veracidad).[24]

Modelado de datos flexible

Todas las bases de datos RDBMS se basan en el modelo de datos relacionales. Por el contrario, cada clase de sistema NoSQL emplea un modelo de datos diferente. Los modelos de datos NoSQL admiten de forma inherente la partición de datos horizontal entre procesadores en un clúster o incluso procesadores en centros de datos. Las aplicaciones RDBMS normalmente funcionan con un esquema de base de datos fijo, aunque el esquema puede evolucionar lentamente. El cambio de esquema es una actividad bien gestionada y la aplicación pasa por rigurosas pruebas de software. Por el contrario, muchos sistemas NoSQL tratan el esquema como una construcción que evoluciona inherentemente con el tiempo. Estos sistemas también requieren un esquema que permita una gran variación en los datos de las filas sin incurrir en problemas de valores nulos. Una aplicación NoSQL trabaja con un modelo de datos que está específicamente diseñado y optimizado para esa aplicación.

Consistencia eventual

Crear, Leer, Actualizar y Eliminar (CRUD, por sus siglas en inglés) son cuatro operaciones que se espera que las bases de datos proporcionen y ejecuten de

manera eficiente. Los modelos de datos NoSQL están estructurados para admitir operaciones masivas de lectura y de inserción simultáneas en relación con las actualizaciones. La noción de consistencia eventual se adopta para admitir operaciones de inserción y lectura extremadamente rápidas. La coherencia eventual implica que las aplicaciones son conscientes del problema de lectura no repetible debido a la latencia en la coherencia. Algunos sistemas NoSQL emplean modelos de datos para admitir de manera eficiente solo insertar y leer con exclusión de la actualización y eliminación.[22]

Escalabilidad horizontal

La escalabilidad horizontal se refiere a la escala mediante la adición de nuevos procesadores completos con sus propios discos (nodos). Otro concepto muy asociado a la escalabilidad horizontal es el particionamiento de datos. Cada nodo / procesador contiene sólo un subconjunto de los datos. El proceso de asignación de datos a cada nodo se denomina fragmentación, que se realiza automáticamente en algunos sistemas NoSQL. Es más fácil lograr escalabilidad horizontal con bases de datos NoSQL. Se reduce considerablemente la complejidad involucrada en hacer cumplir las propiedades ACID. Además, sólo las operaciones de inserción y lectura predominan en las bases de datos NoSQL, debido a que las operaciones de actualización y eliminación se vuelven insignificantes en términos de volumen. Los sistemas NoSQL también logran escalabilidad horizontal al delegar el compromiso del conocido Commit en dos fases (2PC) requerido para la implementación de transacciones distribuidas en las aplicaciones. [23]

2.2.2. Limitaciones

Las bases de datos NoSQL presentan ciertas limitaciones si se las compara con las bases de datos relacionales [21]. A continuación se listan algunas de ellas con el objetivo de ilustrar de que el uso de las mismas no aplica en todos los escenarios:

- No soportan SQL, siendo este el estándar de la industria
- No todas soportan el concepto de transacción
- Al ser una tecnología emergente, muchas de las implementaciones no están lo suficientemente maduras para su uso a nivel comercial

2.2.3. Tecnologías

En la actualidad existen diferentes tipos de bases de datos no relacionales como de clave-valor, en memoria, documentales y basadas en grafos, entre otras. Este proyecto está centrado en las bases de datos documentales y bases de datos basadas en grafos.

Para la elección de los motores a evaluar dentro de cada categoría se realizaron búsquedas en varios sitios. Inicialmente se consideraron sitios como Google

Trends, Stack Overflow y similares para tener un panorama general. Finalmente se utilizó el sitio DB Engines, que considera a éstos y construye rankings teniendo en cuenta su popularidad [25].

En la figura 2.3 se muestran los primeros cinco DBMS que tienen un modelo basado en grafos ubicados en dicho ranking. En la figura 2.4 los primeros 6 para bases de datos documentales. La cantidad de DBMS por categoría se eligió de acuerdo al puntaje relativo entre ellos.

Rank			DBMS	Database Model	Score		
Apr 2021	Mar 2021	Apr 2020			Apr 2021	Mar 2021	Apr 2020
1.	1.	1.	Neo4j	Graph, Multi-model	51.04	-1.28	+0.24
2.	2.	2.	Microsoft Azure Cosmos DB	Multi-model	33.52	+1.11	+1.47
3.	3.	3.	ArangoDB	Multi-model	4.77	-0.28	-0.11
4.	4.	4.	OrientDB	Multi-model	4.23	-0.48	-0.29
5.	5.	5.	Virtuoso	Multi-model	3.18	+0.30	+0.56
6.	↑ 7.	↑ 8.	GraphDB	Multi-model	2.24	-0.03	+1.08
7.	↓ 6.	7.	JanusGraph	Graph	2.10	-0.33	+0.32
8.	8.	↓ 6.	Amazon Neptune	Multi-model	1.90	+0.04	+0.09

Figura 2.3: Ranking de BD de grafos [26]

Rank			DBMS	Database Model	Score		
Apr 2021	Mar 2021	Apr 2020			Apr 2021	Mar 2021	Apr 2020
1.	1.	1.	MongoDB	Document, Multi-model	469.97	+7.58	+31.54
2.	2.	2.	Amazon DynamoDB	Multi-model	70.73	+1.84	+6.46
3.	3.	3.	Microsoft Azure Cosmos DB	Multi-model	33.52	+1.11	+1.47
4.	4.	4.	Couchbase	Document, Multi-model	30.76	-0.24	+0.35
5.	5.	↑ 6.	Firebase Realtime Database	Document	16.92	-0.02	+4.27
6.	6.	↓ 5.	CouchDB	Document, Multi-model	16.01	-0.50	-1.76

Figura 2.4: Ranking de BD Documentales [27]

Como observación, para los motores que aparecen como *Multi-model* se incluyen también en su puntaje el puntaje relacionado a todas sus implementaciones. Por ejemplo, se puede apreciar que Microsoft Azure Cosmos DB tiene el mismo puntaje en ambos listados, ya que tiene implementaciones tanto para bases de datos documentales como basadas en grafos.

Otra fuente para tener en cuenta el crecimiento y utilización de los DBMS NoSQL en la industria fueron los reportes que realiza la empresa Forrester [10]. En la figura 2.5 se presentan las bases de datos basadas en grafos más destacadas en el último trimestre del año 2020. En la figura 2.6 se muestran las bases de datos NoSQL más destacadas de principios de 2019 según otro reporte realizado por Forrester.

Entre las destacadas como líderes o fuertes se eligieron las que cuentan con versiones gratuitas para poder hacer pruebas básicas de conversión de los esquemas, que serán presentadas más adelante.

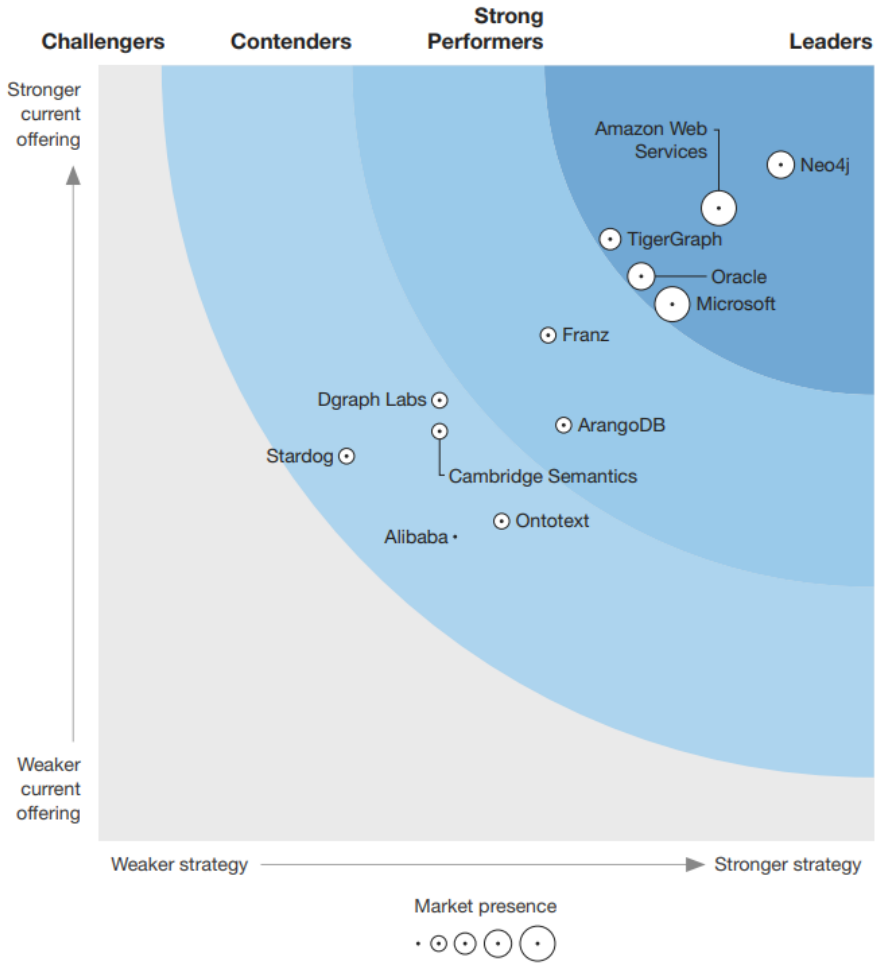

Figura 2.5: Lideres mercado en BD basadas en grafos [28]

Documentales

Una base de datos de documentos o documental es una base no relacional que almacena datos como documentos estructurados, típicamente XML o JSON. Estas bases de datos no son las bases de datos de documentos en el sentido tradicional. Tampoco son sistemas de gestión de contenido. Las bases de datos de documentos se utilizan para administrar datos semi-estructurados principalmente en forma de pares clave-valor empaquetados como documentos JSON. Cada documento es una entidad independiente con atributos potencialmente variados y anidados. Los documentos se indexan por sus identificadores principales, así como por los valores de los campos de documentos semi-estructurados. Las bases

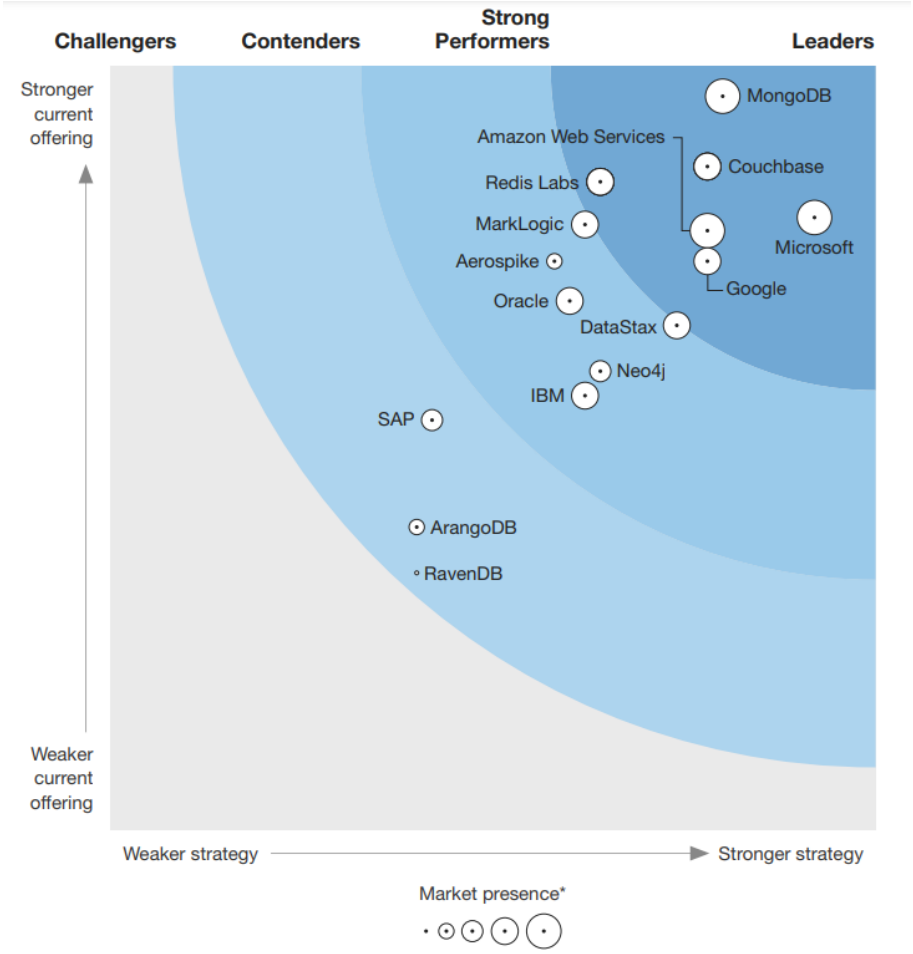

Figura 2.6: Líderes NoSQL DBs [29]

de datos de documentos son ideales para aplicaciones que involucran agregados en colecciones de documentos.

En cuanto al mapeo, el principal problema a resolver es cómo mapear los documentos embebidos en un esquema relacional. En un documento esto puede representar una propiedad del objeto independiente o una relación con documentos en otra colección. Diferenciar entre ambos casos automáticamente no es posible salvo que el manejador implemente mecanismos de clave foránea como por ejemplo el ofrecido por MongoDB [30]. Esto no evita igualmente que puedan existir relaciones definidas sin utilizar esta funcionalidad, lo que dificulta la conversión entre implementaciones.

MongoDB Es el DBMS documental más popular en la industria [29]. Almacena documentos JSON agrupados en colecciones con estructuras flexibles, esto implica que se puede variar la estructura de los documentos a lo largo del tiempo sin que requiera modificar documentos ya existentes. Es una base de datos distribuida en su núcleo, por lo tanto la disponibilidad, escalamiento horizontal y la distribución geográfica están integradas nativamente. Desde la versión 4.0 ofrece además transacciones ACID multi-documento.

Hay aplicaciones utilizando MongoDB en múltiples áreas como *IoT*, *Big Data*, aplicaciones móviles, análisis en tiempo real, entre otras. Además es usada por más de la mitad de las empresas en la lista Fortune 100 [29].

En una base de datos de MongoDB se definen colecciones donde se agregan los documentos que forman parte de la misma. Cada documento es un objeto JSON, por lo que se pueden representar relaciones entre documentos de dos formas distintas: referencias al documento relacionado (listado 1) o documentos embebidos (listado 2). En los listados de ejemplo se presenta una relación que modela jerarquías entre empleados. MongoDB define además una forma explícita de definir relaciones entre documentos con el mecanismo llamado DBRefs [30]. En el listado 3 se muestra cómo se representaría el ejemplo de la jerarquía vista anteriormente.

```
{
    "ci": 1234567,
    "nombre": "empleado X",
    "jefe": 1122345
}
```

Listing 1: Relación como referencia simple

```
{
   "ci":1234567,
   "nombre":"empleado X",
   "jefe":{
      "ci":1122345,
      "nombre":"Jefe A"
   }
}
```

Listing 2: Relación como documento anidado

Mapeo con RDBMS

Se puede realizar una correspondencia entre un esquema de base de datos relacional con una base de datos en MongoDB como se puede ver en la figura 2.7.

```
{
    "ci": 1234567,
    "nombre": "empleado X",
    "jefe":
     {
        "$ref": "empleados",
        "$id" : ObjectId("5126bc054aed4daf9e2ab772")
     }
}
```

Listing 3: Relación como referencia al documento (DBRefs)

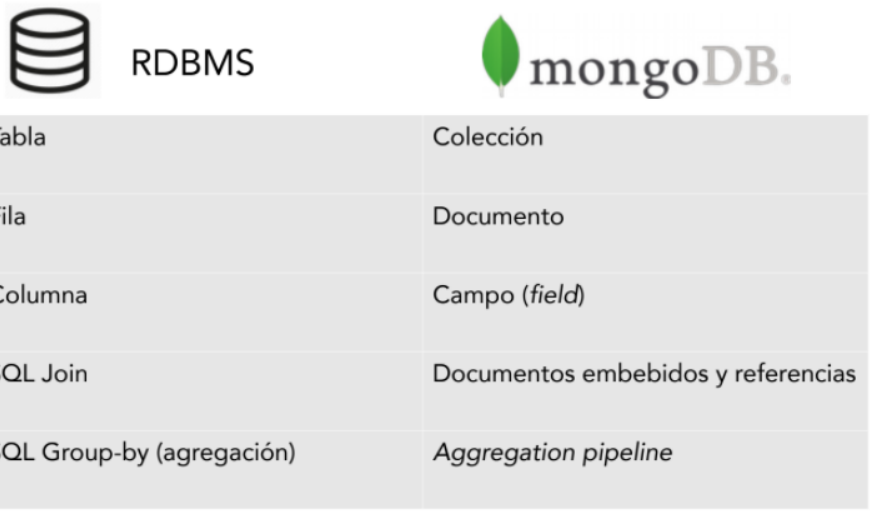

Figura 2.7: Correspondencia RDBMS y MongoDB [31]

Amazon DynamoDB Admite modelos de datos de documentos y clave-valor. Se trata de una base de datos completamente administrada, duradera, multiactiva y de varias regiones que cuenta con copia de seguridad, restauración y seguridad integradas en la nube de Amazon WS.

Provee una API con la cual se puede acceder y manipular la información almacenada en las tablas [32]. Entre las operaciones disponibles se encuentra *DescribeTable* que retorna una estructura de la tabla consultada en formato JSON [33]. Para el mapeo al modelo relacional se podría trabajar directamente sobre esta estructura.

Couchbase Es una base de datos *Open Source* distribuida y multimodelo optimizada para aplicaciones interactivas. Fue diseñada para almacenar documentos y pares clave-valor de forma fácilmente escalable, procesando y accediendo dicha

información con requerimientos de baja latencia.

En cuanto al modelo de datos, también almacena documentos JSON en colecciones. Los documentos pueden ser accedidos a través del lenguaje de consulta N1QL. Específicamente cuenta con la sentencia INFER que retorna un JSON con la estructura de la colección incluyendo cantidad de nodos y propiedades con información de presencia en los documentos [34].

Para desarrollo provee una API en varios lenguajes, Java entre ellos, que brinda acceso a operaciones para el manejo de datos. [35]

Grafos

Un grafo es un conjunto de objetos discretos, llamados nodos, donde cada nodo se conecta con otros a través de relaciones o aristas. Por lo tanto, los componentes de un grafo son:

- Vértices o nodos que representan los distintos objetos
- Aristas o relaciones que conectan los nodos

Los grafos son una forma útil de modelar datos con fines de análisis matemático y han existido durante mucho tiempo. El primer uso de grafos que se tiene registro fue realizado por Euler en 1735, donde usó la teoría de grafos para demostrar que el famoso problema de los Siete Puentes de Konigsberg no tenía solución. [36]

La teoría de grafos proporciona notación matemática para definir nodos, representar relaciones de grafos y realizar operaciones para encontrar nodos adyacentes. Estas operaciones primitivas se pueden utilizar para realizar recorridos en un grafo.

Tanto los nodos como las relaciones pueden tener propiedades o atributos. Las propiedades de los nodos no son diferentes a las que se pueden encontrar asociadas con una tabla relacional o en un documento JSON.

Bases de datos orientadas a grafos Una base de datos orientada a grafos es un sistema de administración y gestión de bases de datos online que provee las operaciones CRUD (Create, Read, Update, Delete) y funcionan bajo un modelo de representación en forma de grafos. Estas bases de datos usualmente se utilizan como parte de un procesamiento de transacciones empresariales online (OLTP, por sus siglas en inglés [37]). En consecuencia, normalmente se optimizan para el rendimiento y se diseñan teniendo en cuenta la integridad transaccional y disponibilidad operativa.

A diferencia de otras bases de datos, las relaciones tienen la máxima prioridad. Esto significa que las aplicaciones no tienen que inferir conexiones de datos usando claves foráneas o procesamiento fuera de banda, como por ejemplo MapReduce. Al representar nodos y relaciones como estructuras directamente conectadas, las bases de datos de grafos permiten construir modelos sofisticados que se mapean muy cercanamente al dominio del problema que se quiere modelar y resolver.

Para aplicaciones de ciencia de datos este tipo de motores son ideales ya que pueden utilizar las relaciones de un modelo para ayudar a empresas, por ejemplo, a tomar decisiones comerciales [38].

En el ejemplo que se muestra en la figura 2.8, se puede ver una instancia de modelo de grafo que representa actores, directores y películas.

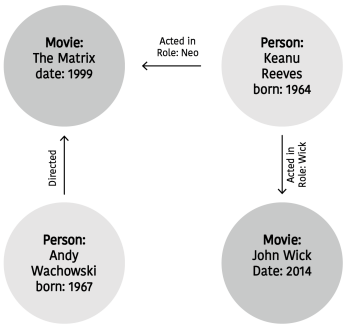

Figura 2.8: Ejemplo de grafo [39]

Se pueden considerar cuatro estructuras de grafos: simples, hipergrafos, anidados y de atributos. La estructura básica es un grafo simple plano definido como un conjunto de nodos o vértices conectados por aristas. Un hipergrafo extiende esta noción permitiendo a una arista relacionarse con un conjunto arbitrario de nodos (a estas aristas se les llama hiper-aristas). Un grafo anidado es un grafo en el cual sus nodos pueden ser grafos en sí mismos (llamados hiper-nodos). Los grafos de atributos son aquellos en los cuales sus nodos y aristas pueden contener atributos para describir sus propiedades. Adicionalmente, sobre estos tipos de grafos, se pueden considerar aristas dirigidas o no, nodos/aristas con o sin etiquetas y con o sin atributos (aristas entre aristas se pueden modelar). [40]

Análisis de motores

Para realizar el análisis de algunos motores de bases de datos orientadas a grafos se tomó como referencia el reporte de Forrester [28] priorizando los motores open source o con versiones *community*. Se hicieron pruebas de con-

cepto sobre los manejadores Neo4j y ArangoDB. Por otro lado, se investigó la documentación de Amazon Neptune y Azure Cosmos DB.

Neo4j Neo4j es una base de datos de código abierto para uso no comercial orientada a grafos que se compone de dos elementos fundamentales: nodos y relaciones. Los nodos representan entidades y las relaciones representan conexiones o interacciones entre los diferentes nodos. Cada uno de estos elementos puede ser etiquetado con un label que es mapeable a una tabla de un esquema relacional. Tanto los nodos como las relaciones pueden contener propiedades. Estas se pueden mapear a columnas de las tablas en un modelo relacional.

Para cada nodo se guardan punteros a los nodos a los que está relacionado y viceversa; con el objetivo de lograr hacer recorridos de k-adyacencia de forma óptima. Es un motor de persistencia Java integrado, totalmente transaccional que almacena datos estructurados en grafos en lugar de tablas. [41]

Neo4j implementa un lenguaje de consulta declarativo para grafos llamado Cypher. Cypher permite consultar grafos utilizando una sintaxis simple algo comparable a SQL, pero especialmente optimizada para recorridos de grafos.

Tiene una API orientada a objetos que permite acceder y manipular la información almacenada mediante una variada cantidad de operaciones. Algunas de ellas, que resultan de especial interés para el proyecto, son las que permiten obtener la metadata y cómo es el esquema de esa base de datos, es decir, los nodos, relaciones, sus propiedades y tipos. A continuación se presentan algunas de las operaciones que expone la API:

```
db.schema.visualization()

db.schema.nodeTypeProperties()

db.schema.relTypeProperties()
```

Como resultado de la investigación y mediante la instalación de un plugin en la base de datos se puede invocar a una operación que devuelve un JSON con la información completa de la metadata y los registros (instancias) que tiene la base de datos sirviendo de alternativa a las operaciones anteriores mencionadas. [42]

```
call apoc.meta.data();
call apoc.meta.schema();
```

Azure Cosmos DB Azure Cosmos DB es un servicio de base de datos multimodelo creado por Microsoft, que ofrece una proyección de API para todos los tipos de modelos NoSQL principales: familia de columnas, documentos, grafos y pares clave-valor. Proporciona una API de Gremlin [43] para aplicaciones que necesitan modelar, consultar y recorrer grandes grafos de manera eficiente. Dentro de los puntos altos de este DBMS se destaca la facilidad de escalado de

la plataforma y la API de consultas múltiples. Los usos que se le dan a Azure Cosmos DB son para aplicaciones transaccionales de misión crítica, data science y grafos de conocimiento, entre otros. [44]

Arango DB Es una base de datos multimodelo open source que combina los modelos no relacionales Clave/Valor, Documentos y Grafos en un solo núcleo, haciendo uso del lenguaje de consulta AQL (ArangoDB Query Language). Este lenguaje permite realizar consultas entre los diferentes modelos de datos indistintamente. La plataforma se puede desplegar localmente y en la nube, incluidos AWS, Google Cloud Platform y Microsoft Azure, combinando el poder de los grafos con documentos JSON, almacenamiento de clave-valor y un motor de búsqueda de texto para permitir que los desarrolladores accedan e integren todos los datos para admitir varias aplicaciones. [45]

Desde el punto de vista del modelo de datos puede considerarse como una base de datos orientada a documentos. La jerarquía es la siguiente: los documentos se agrupan en colecciones y las colecciones existen dentro de la base de datos.

A este modelo se le agrega más estructura para que soporte el almacenamiento y consulta de grafos. Presenta dos tipos de colecciones: colecciones de documentos y colecciones de relaciones. Las colecciones de relaciones (Edges) almacenan documentos y a estos se le incluyen dos atributos especiales: _from y _to. Estos atributos se utilizan para crear relaciones entre documentos esenciales para la conformación de la base de datos de grafos. Las colecciones de documentos también son denominadas colecciones de vértices en el contexto de grafos.

ArangoDB expone su API a través de HTTP, permitiendo que el servidor sea accesible más fácilmente con una variedad de clientes y herramientas [46]. Con esta completa API se puede acceder y manipular la información almacenada. Expone también operaciones para recorrer el grafo (Graph Traversal).

Amazon Neptune Amazon Neptune es un servicio de base de datos basada en grafos totalmente administrada que admite grafos de propiedades y RDF de W3C, aprovechando Apache Tinker Pop Gremlin y SPARQL, lo que ofrece a las organizaciones la capacidad de admitir una amplia gama de casos de uso. Admite conexiones de cliente cifradas con HTTPS y también permite cifrar bases de datos con claves que administra a través de AWS Key Management Service (KMS). Dentro de los motivos que sitúan a Amazon Neptune como uno de los líderes en este tipo de bases de datos se destaca la facilidad de configuración de la plataforma, el hecho de que es parte del ecosistema de AWS, su soporte técnico y su rendimiento.

2.2.4. Resumen

Las bases de datos NoSQL son flexibles en cómo representan y manejan los datos en comparación con las relacionales. Esto genera heterogeneidad entre los

motores debido a que cada uno se enfoca en optimizar algún aspecto del manejo de datos, típicamente lectura y escritura. Esta optimización está orientada a resolver ciertos tipos de escenarios y depende fuertemente de la naturaleza de los datos que maneja. A pesar de esto, la representación interna de la información en los distintos motores no afecta la capacidad de obtener y manejar los datos para cumplir con el objetivo planteado.

Mediante la implementación de la prueba de concepto se pudo verificar que las estructuras que utilizan los motores de bases de datos NoSQL son mapeables a una estructura relacional. Luego de haber investigado algunos motores de bases de datos, todos ellos de alguna forma proveen acceso a todos los datos que tienen almacenados, a partir de los cuales se puede construir un esquema, independientemente de si las interfaces (APIs) que provee el motor brindan alguna operación para obtener el esquema de la instancia.

Consideramos que obtener la estructura a partir de los datos es la estrategia más segura para realizar el mapeo a la base de datos relacional. Para los motores en los cuales hay un esquema definido o la API provee primitivas para inferirlo, esta conversión puede ser más sencilla. De esta forma es que se procedió a realizar las pruebas de concepto ya que los motores de bases de datos utilizados brindan esa posibilidad.

Una consideración importante es que como las bases de datos NoSQL no requieren un esquema, pueden existir muchos valores nulos para las instancias. La dificultad se manifiesta al manipular los datos en el esquema relacional, ya que eventualmente van a existir muchos registros con valores nulos, que afecten la calidad de los datos.

Otro resultado de la investigación realizada es que existen dificultades con las bases de datos NoSQL en cuanto a su estructura, y que se debe analizar individualmente en cada motor. La ausencia de estándares implica que cada motor desarrollado es libre de implementar la forma de acceder a los datos como le sea más conveniente. A pesar de esto, existen intenciones de llegar a un consenso entre la academia y las empresas para lograr estándares. Por ejemplo, para los motores de bases de datos basados en grafos está en proceso la estandarización del lenguaje de consulta GraphQL [47].

Capítulo 3

Análisis del problema

En este capitulo se presenta un análisis en torno a cada uno de los objetivos planteados inicialmente.

Por un lado, en la sección Mapeo se ven el profundidad las formas de representación existentes en MongoDB con énfasis en la identificación de relaciones. Además se presentan los mapeos encontrados entre bases de datos basadas en grafos y relacionales. En ultima instancia se comenta sobre la existencia de esquemas en ambos tipos de bases de datos.

En la sección siguiente, se introduce al metamodelo, factor clave a la hora de realizar un prototipo a modo de prueba de concepto en base a lo analizado en la sección de Mapeo. Aquí también se presentan los algoritmos de carga de dicho metamodelo a alto nivel.

Finalmente se presenta un análisis de la API Genérica BPMS, teniendo en cuenta los datos que se determinaron eran necesarios para cargar el metamodelo. Además se definen los cambios que deben realizarse para tener una carga completa del metamodelo.

3.1. Correspondencia de modelos

Esta etapa consiste en realizar una comparación entre bases de datos relacionales y las bases de datos NoSQL documentales y basadas en grafos con el objetivo de buscar correspondencias y similitudes entre las distintas representaciones. Lo que se busca es poder determinar la viabilidad y los puntos diferentes a tener en consideración a utilizar bases de datos NoSQL. Por otro lado, la investigación de cómo son utilizados los esquemas en las bases de datos NoSQL. Finalmente, se hace una propuesta inicial de lo que se entiende que debe contener la API NoSQL. Se muestra la evolución del trabajo y los cambios realizados en la API respecto a la versión inicial.

3.1.1. Documentos

Una base de datos documental se compone de un conjunto de colecciones. Una colección se compone de un conjunto de documentos. En el caso de MongoDB los documentos son archivos JSON, por lo que un documento es una lista de pares clave-valor, clave string llamada campo y valor de varios tipos posibles (string, numéricos, booleanos, otros documentos, listas de estos tipos, entre otros).

La correspondencia de los conceptos de una base de datos documental con una relacional puede verse en el cuadro 3.1.

Documental	**Relacional**
Colección	Tabla
Documento	Fila
Campo	Columna Relación (clave foránea) Tabla

Cuadro 3.1: Correlación bases documentales y relacionales

¿Que puede representar un campo en un documento JSON?

Campo simple

- puede representar simplemente un atributo de la entidad. Tipos básicos como string, int, bool, etc.. Se corresponde a una columna en un esquema relacional, entonces se mapea directamente con una columna de una tabla
- puede representar una clave foránea (relación 1:1) sobre otra entidad. Se corresponde con una columna en un esquema relacional (clave foránea)

Por ejemplo, los campos CI, Nombre y FechaNacimiento del documento del listado 4 son campos simples de tipo String y Date, respectivamente. El campo Conyuge es una clave que hace referencia a otro documento en la misma colección mientras que el campo LugarDeNacimiento hace referencia a un documento en otra colección.

Campo de tipo arreglo de objetos Distinguimos dos casos, puede representar:

- un atributo multivaluado, como por ejemplo los números de teléfono. Hay 3 variantes para este caso:
  - Serializado
  - Tabla o múltiples columnas (en caso de que sea acotada la cantidad)

```
{
        "CI": "5555666-7",
        "Nombre": "John Doe",
        "FechaNacimiento": "11/05/2000",
        "LugarDeNacimiento": 12,
        "Conyuge": "1234567-8"
}
```

Listing 4: Campos simples

- Una relación 1:N hacia otra entidad en la misma colección u otra colección (al estilo clave foránea), entonces se crea una tabla para la relación

```
{
    "CI": "5555666-7",
        "Nombre": "John Doe",
        "FechaNacimiento": "11/05/2000",
        "LugarDeNacimiento": 12,
        "Telefonos": [ 123134, 14124214 ],
        "Conyuge": "1234567-8",
        "Hijos": [ "6666657-7", "3546511-3" ]
}
```

Listing 5: Campos de tipo arreglo

El campo Teléfonos del ejemplo del listado 5 es un atributo multivaluado, mientras que el campo Hijos representa una relación sobre la misma colección de personas.

Campo objeto JSON Hay dos casos, puede representar:

- Un tipo de dato complejo al estilo *data type* (como por ejemplo una dirección que se compone de dos subcampos calle y número de puerta) entonces se puede mantener como un único objeto serializado pero puede generar problemas a la hora de consultar o actualizarlo. Hay otras alternativas que consisten en crear un atributo simple para cada subcampo del atributo o crear una tabla nueva para representar esta relación.
- Ser un documento de otra colección o la misma colección (embebido), como relación por copia; se guarda una copia de otro documento. Entonces se pasa como tabla que relaciona las entidades

Por ejemplo, el atributo *Direccion* del listado 6 es un caso de atributo complejo, mientras que el campo Trabajo incluye información de otras colecciones (la colección de Empresas y la de Trabajos).

```
{
        "CI": "5555666-7",
        "Nombre": "John Doe",
        "FechaNacimiento": "11/05/2000",
        "LugarDeNacimiento": 12,
        "Telefonos": [ 123134, 14124214 ],
        "Direccion" :
        {
                "Calle": "Comodoro Coe",
                "NumeroDePuerta": "5456",
                "Apartamento": "2B"
        },
        "Conyuge": "1234567-8",
        "Hijos": [ "6666657-7", "3546511-3" ],
        "Trabajo":
        {
                "IdEmpresa": 65465,
                "NombreEmpresa": "Empresa X",
                "Salario": "34141"
        }
}
```

Listing 6: Campos de tipo sub-documento

En todos estos casos puede resultar complejo el determinar si un campo es una clave foránea y, si lo es, de cuál colección o incluso identificar el campo en la colección al cual hace referencia. MongoDB cuenta con un mecanismo que permite eliminar esa ambigüedad si se utiliza adecuadamente [30].

Identificación de relaciones

Un caso particular de las bases de datos documentales es que es complicado poder identificar relaciones y, para este proyecto en particular poder referenciarlas entre distintas consultas a la API. Esto se debe a que las relaciones no se guardan explícitamente sino que deben ser descubiertas analizando todos los documentos de las colecciones de una base de datos. Dada la naturaleza de los casos de uso de este tipo de bases de datos, esto puede consumir una gran cantidad de cómputo.

Para abordar ese problema hay que definir una forma de poder referirse de forma única a una relación encontrada de acuerdo al dominio del problema. Sabemos que en una base de datos de Mongo las colecciones tienen nombre único y también los campos dentro de cada documento de las mismas. La resolución del problema se reduce a encontrar una forma de generar el nombre de la relación concatenando el nombre de la colección origen de la relación y el nombre del campo. Para asegurar que no se construye un nombre de entidad que ya existe

en la base de datos organizacional el carácter unificador debe ser uno que no este permitido en el nombre de una colección en mongo, por ejemplo un punto.

Para detectar relaciones entre documentos es fundamental poder determinar cuándo un campo de un documento está haciendo referencia a otro documento en otra colección, es decir, ese campo es una clave foránea. Paralelamente, que un campo sea clave foránea, implica que es clave primaria de alguna colección.

Yendo a MongoDB, salvo que la DB use el mecanismo de DBRefs, no es directo determinar con certeza las claves primarias y foráneas de una base de datos. Resumiendo entonces, el problema se reduce a detectar claves primarias para luego detectar las claves foráneas que manifiestan la existencia de relaciones entre documentos. Es por esto que analizamos distintas alternativas que se detallan a continuación.

DBRefs Ventajas:

- Permiten identificar unívocamente a qué colección pertenece la referencia del campo en cuestión

Desventaja:

- Parecen ser poco usadas, potencialmente no se encuentran en ninguna base real. Su uso se recomienda en escenarios específicos. Sería una utilidad casi teórica, de construir DBs para probar la carga y no de cargar una base real con la API.

Descubrimiento del esquema automáticamente Esto implica descubrir totalmente el esquema de la DB para poder determinar claves primarias y foráneas de cada colección.

Ventaja:

- Funcionaria para cualquier base de datos

Desventajas:

- Costoso computacionalmente porque para cada operación de la API NoSQL potencialmente requiere recorrer todos los documentos de una o más colecciones para detectar relaciones

- Propenso a errores de identificación de claves por pocos datos o datos "sucios". Por ejemplo, puede haber un campo que tenga valores únicos (nombres de personas) pero en realidad no es clave primaria. En este caso es imposible automáticamente determinar que no es una clave de la colección, debido a que en MongoDB no existe el concepto de clave primaria definida por el usuario (sí existe una generada automáticamente).

Índices La segunda desventaja de la alternativa anterior, que la hace prácticamente inviable, es posible de remediar si la base de datos organizacional cuenta con índices de unicidad sobre las claves de la realidad de cada colección [48]. Por ejemplo, para las personas es la cédula de identidad.

Los índices se utilizan sobre las claves primarias para agilizar las consultas sobre las colecciones. También es posible usarlos en campos que no sean claves primarias, en ese caso no tienen la propiedad *unique* en true.

Si la base de datos usa índices para sus claves primarias podría mejorar el rendimiento del algoritmo de carga del metamodelo. Es algo que todas las bases deberían tener dada la magnitud de datos que se suelen almacenar en las bases documentales.

Esquema Requiere que la base ya tenga un esquema previo definido, esto es posible realizar en MongoDB [49]. Estos esquemas permiten definir campos requeridos, lo cual es útil para acotar el conjunto posible de campos que pueden ser clave primaria de una colección.

Información de claves primarias y foráneas como entrada Consiste en que el cliente que usa la API provea una estructura de datos donde brinda información de las claves foráneas de cada colección. Un ejemplo de estructura podría ser el que muestra en el listado 7.

```
[{
        "Collection": "Personas",
        "Keys": ["CI"],
        "ForeignKeys": [
        {
                "Field": "NacioEn",
                "TargetColection": "Paises",
                "TargetField": "CodigoISO"
       },
       {
                "Field": "TrabajaEn",
                "TargetColection": "Empresas",
                "TargetField": "IdEmpresa"
       }
       ]
}]
```

Listing 7: Ejemplo de esquema de entrada

Ventajas:

- Permite ahorrar tiempo de cómputo
- Se eliminan ambigüedades y posibles errores de identificación de claves primarias y foráneas

- Funciona para cualquier base de datos ya sea que use o no índices, DBRefs o esquemas

Desventajas:

- Puede no ser conocido por el usuario que hace la consulta

3.1.2. Grafos

Una base de datos basada en grafos se compone de nodos y aristas que relacionan nodos entre sí. Los nodos pueden tener etiquetas que los diferencian entre sí, al igual que las aristas. Tanto nodos como aristas pueden tener atributos. Las relaciones son representadas a través de las aristas. En el cuadro 3.2 se presenta la correspondencia entre los conceptos de las bases de datos de grafos y los conceptos en un esquema relacional tradicional.

Grafos	**Relacional**
Etiqueta en Nodo / Arista	Tabla
Nodo / Arista	Fila
Propiedad	Columna
Aristas entre nodos	Claves foráneas

Cuadro 3.2: Correlación bases de grafos y relacionales

3.1.3. Esquemas en bases de datos NoSQL

A efectos de comparar como es el funcionamiento de los esquemas de bases de datos relacionales y mapear ese comportamiento a bases de datos basadas en grafos se define a un esquema como un conjunto de restricciones proporcionadas por el usuario final al DBMS con la expectativa de que el DBMS imponga estas restricciones con el objetivo de mantener la integridad de los datos. Los esquemas en el mundo relacional incluyen tablas, vistas, índices, claves foráneas y restricciones.

Esquemas en bases documentales

MongoDB permite definir un validador al momento de crear una colección (o sobre una ya existente) con el objetivo de definir de esta forma reglas que se validan al momento de intentar agregar o modificar documentos de la colección en cuestión. Este documento validador es un objeto JSON que permite definir los nombres de los atributos permitidos, cuales de ellos son requeridos, tipo de cada atributo y rangos de valores para los mismos, así como también un mensaje para mostrar cuando se intenta ingresar un valor invalido para algún campo, entre otras opciones. [49] [50]

Es configurable también qué tan estricto se comporta respecto a cuando detecta una violación del esquema definido, puede hacer fallar la operación o simplemente levantar una advertencia y permitir la invalidez del documento.

Esquemas en bases de grafos

Para las bases de datos basadas en grafos se incluyen etiquetas de vértice y borde, las *keys* de propiedad, índices y otras variadas opciones de restricciones. Puntualmente en Neo4j la definición de esquemas es opcional. Cuando se usa, su propósito principal es definir índices que están vinculados a etiquetas de vértices específicos y para la especificación de una serie de otras restricciones. Las etiquetas de vértices se crean implícitamente a medida que se agregan datos al sistema. Lo mismo ocurre con las propiedades y las etiquetas de borde (tipos de relación). Las propiedades no se escriben explícitamente, por lo que puede insertar cualquiera de los tipos admitidos. Los índices se pueden vincular a etiquetas de vértices específicos. Por ejemplo, es posible indexar por nombre y edad restringiendo esto a los vértices marcados con una etiqueta específica. Se admite la indexación de propiedades únicas y múltiples (compuesta). Los índices también se pueden eliminar después de que se hayan creado si ya no son necesarios. Neo4j ofrece distintas opciones de tipos de restricciones que tienen como objetivo mantener la integridad de los datos. Muchas de ellas pueden ser utilizadas en la versión *Enterprise*. A modo de ejemplo, la restricciones que chequea que un nodo debe tener una propiedad específica. Esta restricción asegura la existencia de una propiedad para todos los nodos dada una etiqueta. Las consultas que intentan crear nuevos nodos de la etiqueta especificada, pero sin esta propiedad, fallarán. Lo mismo ocurre con las consultas que intentan eliminar una propiedad marcada como obligatoria. [51] [52]

3.2. API NoSQL

En esta sección se presenta al metamodelo a nivel conceptual, ya que los datos que se deben extraer de las bases de datos organizacionales están determinados por los datos que el metamodelo requiere. Luego de presentarse los aspectos principales del metamodelo, se analizan las operaciones necesarias para extraer la información.

3.2.1. Metamodelo

El metamodelo es un esquema que pretende poder representar cualquier concepto existente tanto de BPMS como en cualquier base de datos organizacional [6]. En el se representan entidades claves de los BPMSs como procesos, tareas y variables, así como también entidades, atributos y relaciones de bases de datos organizacionales. Además, mediante la ejecución de un algoritmo de matching existente e implementado en [9], se conectan ambas partes del modelo, siendo capaz de identificarse los datos que provienen de la ejecución de un proceso con su respectivo valor en las bases de datos organizacionales. Esto es un factor clave para el posterior procesamiento de la información extraída. Dicho algoritmo es quien conecta la parte superior con la inferior del metamodelo, en caso de detectar que corresponde hacerlo.

Como se puede ver en la figura 3.1, el metamodelo está organizado en 2 partes principales: la superior, que modela todo lo relativo a BPMS; y la inferior que modela todo lo que es datos organizacionales. A su vez, cada una de estas secciones, su subdividen en definiciones e instancias.

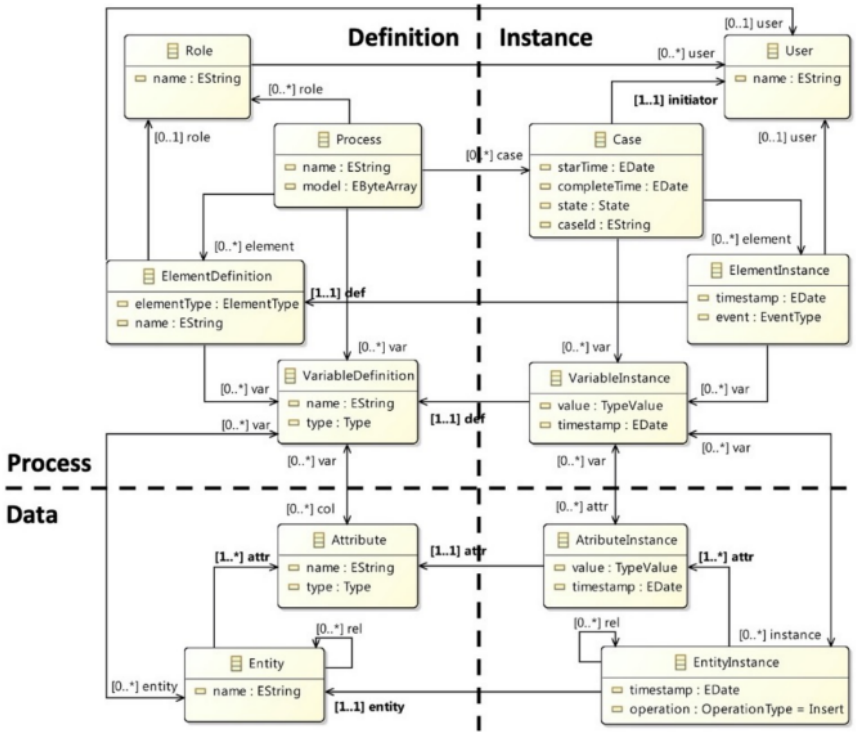

Figura 3.1: Metamodelo inicial [6]

3.2.2. Prueba de concepto

Con el objetivo de realizar un esquema de cómo se pueden mapear modelos en distintos tipos de DBMS se realiza una prueba de concepto básica. Para la misma se usa el esquema conceptual presentado en 3.2.Se utilizó el motor MongoDB para la base de datos documental y Neo4j para la base de datos basada en grafos.

Esquema relacional En este caso se modela con tres tablas (ver 3.3). La relación Trabaja se modela con claves foráneas de las tablas de Empleados y Empresas.

Figura 3.2: Esquema conceptual

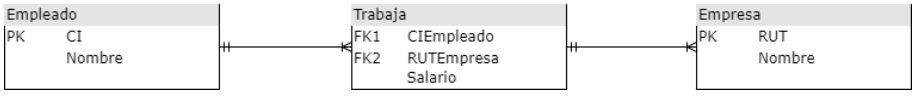

Figura 3.3: Esquema relacional

Esquema documental Se tienen dos colecciones (ver 3.4). Por un lado las empresas con su información. Por otro lado, la colección de empleados que tienen embebida la relación trabaja porque en el modelo pueden tener un único trabajo. Es embebido parcialmente ya que contienen solamente la referencia a la empresa para la cual trabajan. Otra alternativa es tener atributos que interesan para optimizar consultas si el escenario lo requiere.

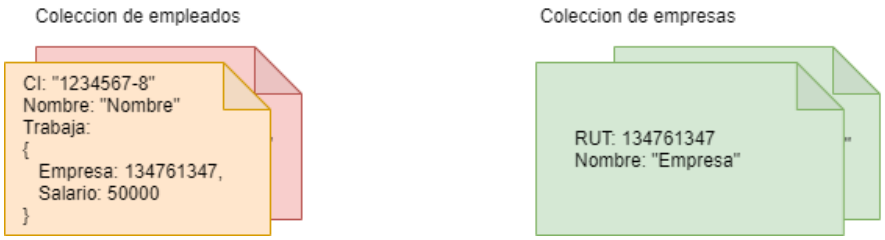

Figura 3.4: Esquema documental

Esquema tipo grafo En un esquema de grafos se tienen nodos con 2 etiquetas distintas 3.5, una para personas (Empleado), otra para las empresas y una arista para la relación Trabaja que vincula a los empleados con las empresas.

Prueba de transformación a relacional Siguiendo el esquema presentado en 4.4, por ejemplo, al invocarse la operación GetEntity("Empleado") en el contexto de la base de datos implementada con neo4j, el módulo se encarga de realizar la conexión con la base, realizar la consulta y retornar en un formato estándar la información a la NoSQL API, la cual envía los datos recuperados al consumidor. Estos datos incluyen la estructura de la entidad Empleado, sus atributos y las instancias de los mismos.

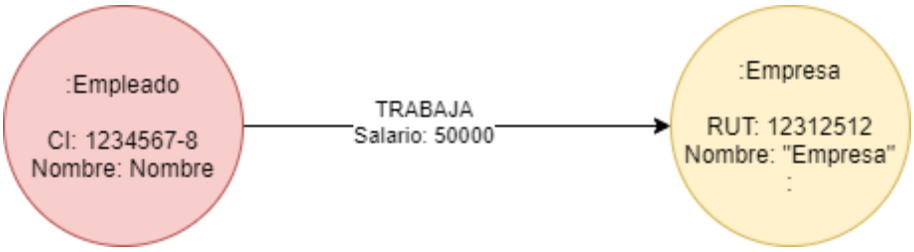

Figura 3.5: Esquema de grafos

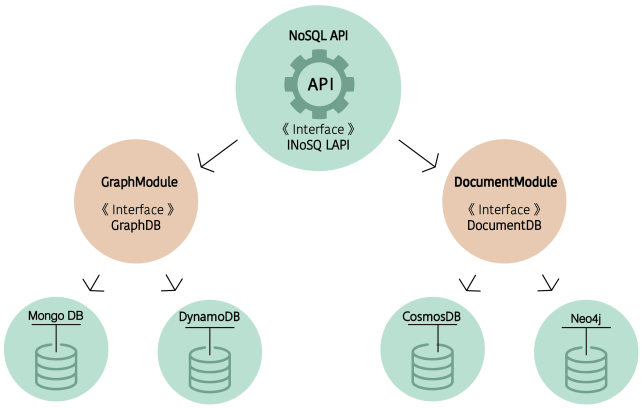

Figura 3.6: Arquitectura de la prueba

Cada módulo implementa su propia lógica de traducción de las respuestas brindadas por las APIs de los DBMS particulares a la aceptada por la API NoSQL. La prueba de concepto se realizó en java para los motores MongoDB y Neo4j. [53]

3.2.3. Versión inicial de la API

La primera versión de la NoSQL API se comenzó a construir tomando como referencia la prueba de concepto. En esta instancia se planteó con sólo una operación teniendo la siguiente firma:

Data getDBEntities(Context context)

Como input para esta operación solamente se solicita la información llamada "contexto" con las credenciales de autenticación para la base de datos organizacional. Este método es instanciado por cada tipo de bases de datos mediante un cliente, el cual conoce la forma de representación de ese tipo de base de datos para obtener la información necesaria. Los datos obtenidos de la base de datos organizacional son entidades, atributos e instancias de ambos. El valor de salida de este método contiene toda esa información. Al tener en un tipo de datos estructurado todo lo necesario de la base de datos organizacional solamente resta pasar esa información capturada al metamodelo.

Para una segunda versión de esta API se separa ese único método ofrecido en varios más pequeños brindando mayor granularidad en la recuperación de la información. La modificación más importante entre las distintas versiones es que todos los métodos ahora se exponen para ser consumidos dejando de estar encapsulado todo en una operación. A continuación se presentan los métodos que son considerados necesarios para dar soporte a la carga del metamodelo.

Se debe resolver el manejo de la conexión, por en caso de servicios Rest mediante token de sesión, de manera que se requiera una única vez enviar los datos de conexión. En caso de que sea una API como dependencia se mantiene la referencia en memoria.

List<String>GetEntityNames()
Retorna una lista con los nombres de las entidades existentes.

List<Entity>GetEntityDefinitions()
Retorna la lista de entidades, incluyendo sus atributos.

Entity GetEntityDefinition(string entityName)
Retorna la información de la entidad dada por su entityName Si la entidad no existe se retorna NULL.

List<EntityInstance>GetEntityInstances(string entityName)
Retorna todas las instancias pertenecientes a una entidad dado por su nombre Si la entidad no existe se retorna NULL.

En base al metamodelo inicial, también se contaban con estructuras y operaciones para cargar relaciones entre entidades, llamadas EntityEntity, pero tenían el inconveniente que no modelaban relaciones con atributos. Por esto se realizó un rediseño del metamodelo incluyendo la tabla Relation y por consiguiente la tabla RelationInstance. Por esto se agregaron las siguientes operaciones:

List<String>GetRelationNames()
Retorna una lista con los nombres de las relaciones existentes.

List<Relation>GetEntityDefinitions()
Retorna la lista de relaciones, incluyendo sus atributos.

Entity GetEntityDefinition(string entityName)
Retorna la información de la relación dada por su relationName Si la relación no existe se retorna NULL.

List<RelationInstance>GetERelationInstances(string relationName)
Retorna todas las instancias pertenecientes a una relación dada por su nombre Si la relación no existe se retorna NULL.

3.2.4. Auditorías

Las organizaciones utilizan diferentes tipos de bases de datos que son impactadas durante la ejecución de los procesos de negocio y generan registros muy valiosos para análisis posteriores. En investigaciones anteriores [9], enfocado en organizaciones donde el tipo de base de datos es relacional, una solución muy útil provista por estos manejadores, puntualmente PostgreSQL, es obtener el registro completo histórico mediante logs. Analizando un poco el contexto, y teniendo siempre como referencia que lo que se debe lograr es que mediante un mecanismo de matching dado, se puedan encontrar correspondencias entre los datos generados por la ejecución de procesos de negocio y las bases de datos organizacionales. Con este fin es que se deben analizar alternativas para abordar este problema, utilizando auditorías en la base de datos organizacional, en este caso, bases NoSQL. Esto habitualmente se logra mediante el registro y acceso a información de tipo de auditorías en los motores de bases de datos, los cuales registran todos los cambios realizados a las entidades de cada base. Los cambios de interés para este proyecto son la marca de tiempo, el tipo de operación y los datos involucrados en la operación (atributos que cambian y sus valores pre y post-operación). Es claro que esta característica es específica de cada DBMS y debe ser estudiada en particular para cada uno.

3.2.5. Carga al metamodelo

Realizamos un prototipo que dada una estructura interna que se lee desde los DBMS noSQL organizacionales permite crear dichas instancias en la base de datos relacional del metamodelo. Este prototipo carga para cada entidad que identifica en la base de datos organizacional de origen sus atributos e instancias en el mencionado metamodelo, considerando la estructura del mismo como claves foráneas y tipos de datos definidos en él.

El algoritmo general consiste en ir insertando en la base las entidades sin dependencias y luego las que tienen sus dependencias (claves foráneas) ya insertadas. La secuencia es Entity, Relation, Attribute, EntityInstance, RelationInstance, AttributeInstance.

Algoritmo de carga

En la API hay definidas operaciones para distintos escenarios. El algoritmo más simple para la carga de las entidades y sus instancias en el metamodelo es el que se muestra en el listado 9.

Hay varios detalles que no se especifican aquí pero se tomaron en cuenta en la versión presentada anteriormente en la prueba de concepto en GitLab, como el manejo de claves foráneas para asociar las instancias creadas y sus atributos. Este aspecto es sumamente importante a la hora de hacer la carga de los datos en el metamodelo.

3.3. API Genérica BPMS

Como vimos en la sección del metamodelo, la parte superior del mismo es la que modela los datos relacionados a procesos de negocio. Estos datos incluyen procesos, tareas y variables, tanto definiciones como instancias de los mismos, principalmente. Por otro lado, se cuenta con la API Genérica para interactuar con distintos BPMS [8]. En esta sección se hace una revisión de las operaciones que esta API provee con el objetivo de seleccionar aquellas que son útiles para la extracción de datos y también determinar qué operaciones es necesario agregar o modificar.

3.3.1. Operaciones disponibles

El primer paso es hacer una revisión tanto de las operaciones que expone la API así como las estructuras de datos que retornan dichas operaciones, de forma de determinar cuales deben ser modificadas y cuales es necesario implementar de cero.

Notar que esta API no fue creada con el objetivo de proveer de datos el metamodelo, sino que se creo para acceder de forma centralizada a distintos BPMS, por lo cual contienen muchas operaciones para modificar procesos e interactuar con ellos. Recordemos que para la carga del metamodelo solamente se lee información, sin afectar el estado de los datos almacenados en los sistemas BPM.

A continuación se listan las operaciones más importantes que son de utilidad para la obtención de los datos de los motores de procesos.

- getUsers
- getGroups
- getTasks
- getProcesses

A su vez, debido a las necesidades de este caso, se determina que es necesario realizar implementaciones de operaciones para el motor Camunda, que inicialmente no se realizaron. Estas son:

```
dbMetamodelo = getTargetDB()

List<String> entitiesStr = getEntityNames()

foreach entityStr in entitiesStr
        Entity entity = getEntityDefinition(entityStr)

    dbMetamodelo.entity.insert(entity)
        foreach attribute in entity.Attributes
                dbMetamodelo.attribute.insert(attribute)
        endfor

        List<EntityInstance> instances =  GetEntityInstances(entity.Name)
        foreach instance in instances
        dbMetamodelo.entityInstance.insert(instance)
        foreach attributeInstance in instance.attributes
                dbMetamodelo.attributeInstance.insert(attributeInstance)
        endfor
    endfor
endfor

List<String> relationsStr = getRelationNames()

foreach relationStr in relationsStr
        Relation relation = getRelationDefinition(relationStr )

    dbMetamodelo.relation.insert(relation)
        foreach attribute in relation.Attributes
                dbMetamodelo.attribute.insert(attribute)
        endfor

        List<RelationInstance> instances =  GetRelationInstances(relation.Name)
        foreach instance in instances
        dbMetamodelo.relationInstance.insert(instance)
        foreach attributeInstance in instance.attributes
                dbMetamodelo.attributeInstance.insert(attributeInstance)
        endfor
    endfor
endfor
```

Listing 8: Algoritmo de carga al metamodelo inferior

- *getGroups*: La operación getGroups se agrega en la definición de la API. La implementación de dicha operación se realiza únicamente para el motor Camunda, dejando como trabajo futuro las implementaciones de los

```
foreach entityName : getEntityNames()
    insertEntity(getEntityDefinition(entityName))
    insertEntityInstances(getEntityInstances(entityName))
endfor
foreach relationName : getRelationNames()
    insertRelation(getRelationDefinition(relationName))
    insertRelationInstances(getRelationInstances(relationName))
endfor


insertUsers(getUsers())
insertRoles(getRoles())
foreach process : getProcesses()
    insertProcess(process)
    insertVariables(process.variables)
    insertTaskDefinitions(process.taskDefinitions)
    insertCases(getCases(process.id))
endfor
```

Listing 9: Algoritmo de carga al metamodelo inferior

restantes motores.

- *getUsersGroup*: Dado un grupo retorna la lista de usuarios que pertenecen a él.
- *getURLMotor*: Retorna la URL del motor de donde provienen los datos de procesos.

Capítulo 4

Solución

En este capítulo se presenta la solución al problema planteado. Se definen y detallan cada uno de los componentes creados en las distintas etapas para dar soporte al proceso de carga del metamodelo utilizando diferentes fuentes de información. En este proceso se aprovecharon y reutilizaron algunos componentes que fueron definidos en proyectos anteriores. En algunos de ellos, como la API genérica, se deben realizar adaptaciones debido a que la misma no fue creada inicialmente para que sea utilizada por un algoritmo de carga del metamodelo. Por otro lado, también se aprovechan conocimientos de investigaciones anteriores donde se procede a la carga del metamodelo utilizando el motor de procesos Activiti. Para el presente proyecto se utiliza otro motor de procesos, Camunda, pero sirve como guía debido a las grandes similitudes entre ambos motores.

El objetivo principal en este caso es desacoplar la carga de un motor particular a la base de datos del metamodelo, es decir, no utilizar la Base de datos del motor de procesos o directamente su API expuesta, sino que realizar la obtención de datos y carga de la información necesaria a través de la API genérica.

El esquema con el que se parte en este proyecto queda ilustrado en la figura 4.1. Representan a los dos proyectos que se toman como punto de partida. Ambos tienen en común que trabajan sobre motores BPMS pero no existe una conexión directa entre sí. Por un lado la API genérica y por otro lado, la carga del metamodelo utilizando el motor de Activiti, donde el proceso de carga se realiza directamente con operaciones a la base de datos relacional de Activiti.

En la figura 4.2 se muestra, en forma esquemática, el objetivo del proyecto. En las siguientes secciones del capítulo se definirán cada uno de los componentes representados en la figura. El código de colores indica que los componentes que están en verde son implementaciones por completo de este proyecto. Por otro lado, los componentes amarillos son los que quedan por fuera de la implementación de este proyecto. Finalmente los azules hacen referencia a componentes ya existentes, pero que se le realizan algunas adaptaciones para que funcionen adecuadamente en el flujo completo.

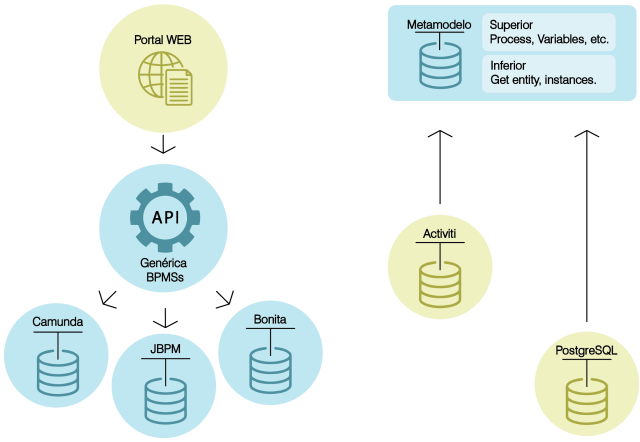

Figura 4.1: Proyectos legados

4.1. Organizational DB Reader

El DBOrgReader es un componente creado con el objetivo de obtener los datos de bases de datos organizacionales para cargar en el metamodelo. Para este caso la idea inicial consistió en generar una API general que pueda aplicarse y escalar a cualquier tipo de bases de datos NoSQL ya que las entidades y componentes a cargar al metamodelo ya se saben de antemano. Para verificar esta idea se procedió a realizar pruebas de concepto con los tipos de bases de datos seleccionadas producto de la investigación de los motores más populares, es decir, Neo4j y MongoDB.Con la validación la prueba de concepto y en base a los resultados obtenidos se decidió pasar a la fase de implementación de un componente que fuera capaz de consultar bases de datos documentales y de grafos de forma genérica exponiendo una API común.

En la figura 4.6 se puede observar, en contexto, lo que abarca la implementación de este componente, estando fuertemente acoplado a lo que es la definición de la API No SQL.

4.1.1. Propuesta de prototipo

Se provee soporte para los DBMS Neo4j de la familia de grafos y MongoDB para la familia de documentales.

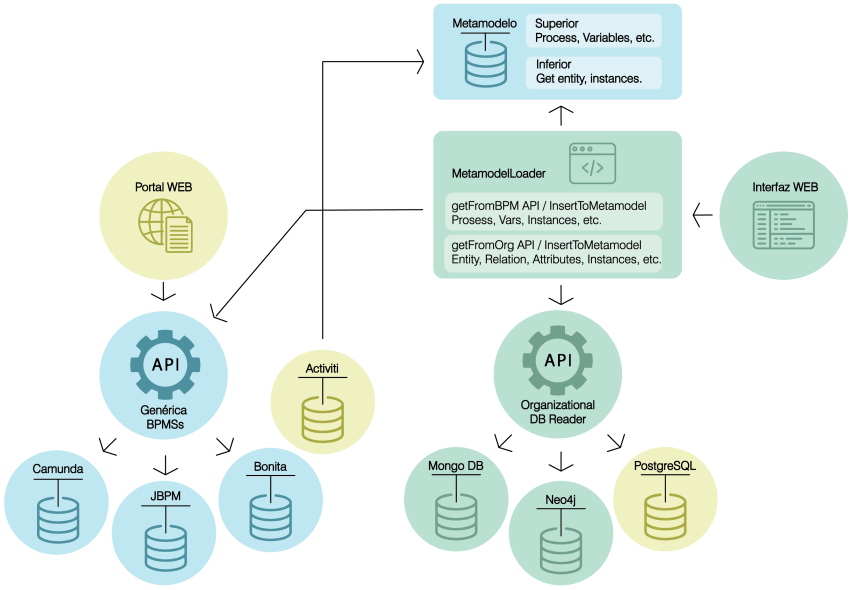

Figura 4.2: Esquema general

MongoDB Para MongoDB soportar referencias con el mecanismo DBRef provisto que permite saber si es una referencia externa indicando la colección destino. De esta forma es inmediato saber si un campo de un documento representa una relación. Adicionalmente, se tiene la restricción de que deben ser referencias siempre sobre la misma base de datos un campo DBRef.

Neo4j Para Neo4j soportar cualquier tipo de bases de datos que hayan sido desarrolladas con este motor.

Arquitectura

Inicialmente se define un esquema con la arquitectura descrita en la figura 4.4. Sin embargo, luego de haber realizado la prueba de concepto y la investigación de distintas herramientas, se constató que no existe un lenguaje estándar para consultar distintos motores de bases de datos. Además, como los distintos tipos de bases de datos, en este caso documentales y basadas en grafos, no presentan una representación interna común, consideramos que tener una capa intermedia para cada tipo de base de datos en la arquitectura no está justificada y se decidió suprimirla, resultando la arquitectura presentada en la figura 4.5.

Esto implica que cada componente que implementa la interfaz de la INoSQLAPI realiza la conversión requerida de acuerdo a la estructura del manejador y los métodos que este ofrece para recuperar la información que sus bases de

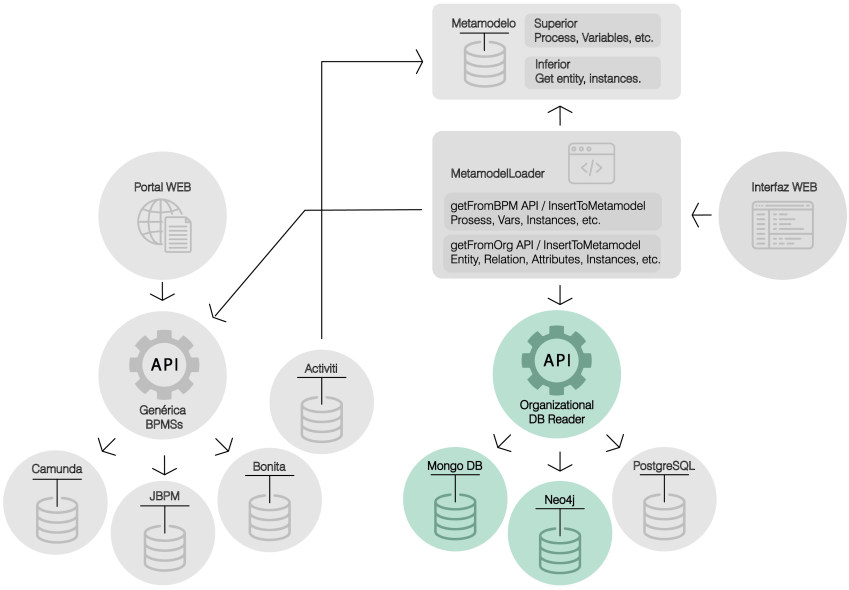

Figura 4.3: Organizational API

datos almacenan.

4.1.2. Auditorías

En línea con lo definido en el capítulo anterior, se deben utilizar auditorías en la base de datos no organizacional y se deben tener presente al momento de extraer los registros de estas bases de datos. A continuación se plantean las tres posibles soluciones planteadas para los manejadores de bases de datos seleccionados, estas son Auditorías generales que se aplican a nivel de código, investigaciones para Neo4j y para MongoDB.

Auditorías por código

Una de las opciones que siempre tiene solución para este tipo de problemas es la inclusión de logs manualmente en el código de la aplicación en las distintas bases de datos no relacionales. Este mecanismo tiene la limitación de que se debe programar previamente en cada aplicación y base de datos, pero a la vez es un recurso útil que se puede aplicar en todas los tipos de bases de datos en caso que se necesite. Para este escenario, una posible implementación de esta solución es agregar en cada entidad que se ingresa a la base de datos un campo timestamp, indicando cuando ese registro se agregó a la Base de datos. Otra posible implementación es tener una estructura dentro de la aplicación

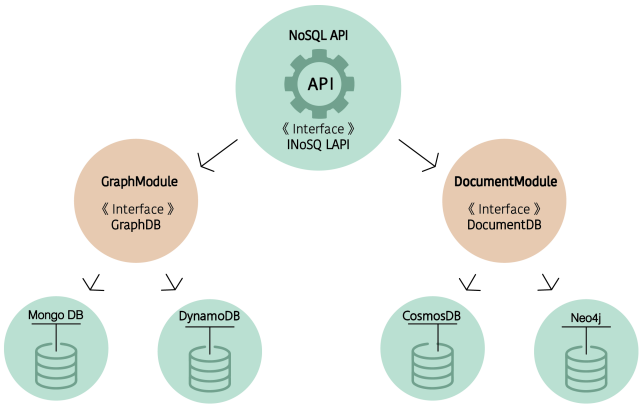

Figura 4.4: Arquitectura inicial

especialmente diseñada para ingresar los logs. Esta opción es la utilizada para el caso de estudio que se presenta mas adelante.

Auditorías en Neo4j

La documentación oficial de Neo4j indica que existe el llamado transaction log que es el encargado de registrar todas las operaciones de escritura en la base de datos [6]. El archivo generado mediante este proceso presenta un alto grado de complejidad al abrir e interpretar, ya que posee una extensión de respaldo de base de datos. Dentro de los archivos generados también hay un log general, llamado *query.log* que muestra los cambios en la base de datos, pero está embebida dentro de un montón de información extra logueada que dificulta el proceso de extracción y lectura automática. La investigación en esta linea se centró en ajustar la parametrización de ese log de forma tal que registre la mínima información posible que sea de utilidad. En una primera instancia al dejar el parámetro de la siguiente manera: dbms.logs.query.enabled=INFO

Este log registra en el archivo cada vez que se escribe un carácter en la consola donde se genera la consulta.

Mirando en detalle cada línea insertada en el log, se puede notar que el *insert* propiamente dicho de la base de datos (create del nodo) tiene un campo llamado type, referido al tipo de la instrucción. En el caso que se loguea, a medida que se tipean caracteres tiene el valor de “user-action”. En la creación del nodo tiene el

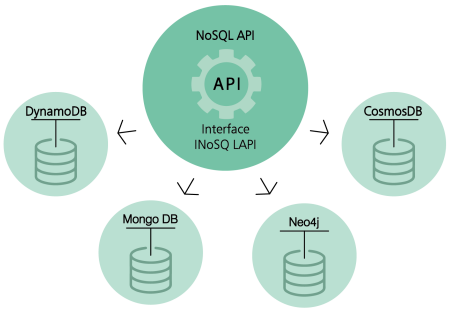

Figura 4.5: Nueva arquitectura

valor de "user-direct". El log presentado anteriormente es un ejemplo resumido en el que se resaltaron los puntos claves en donde se puede trabajar y extraer los datos útiles, por ejemplo, mediante expresiones regulares.

Auditorías en MongoDB

MongoDB cuenta con funcionalidades de auditoría en distintos niveles. En cuanto a las operaciones CRUD, que es lo que interesa obtener, esto está disponible para las versiones enterprise ya que se activa a través de una operación disponible solamente para sus versiones pagas [54].

Para lo que se necesita para el proyecto es necesario que se cree una colección nueva que se actualice cada vez que se realice una operación sobre la base. Un esquema posible puede ser el que se muestra en el listado 10.

Es importante tener en cuenta que la fecha de creación de cada documento está disponible como fue utilizada en la prueba de concepto realizada; forma parte del campo _id que se crea para cada documento agregado a las colecciones. Igualmente, esto es así si no se usa el campo _id para almacenar un valor propio de la realidad, lo cual es posible hacer.

```
{
   "operation":{
      "type":"String",
      "required":true
   },
  {
   "database":{
      "type":"String",
      "required":true
   },
 {
   "collection":{
      "type":"String",
      "required":true
   },
   "newValues":{
      "type":"Object",
      "required":false
   },
   "timestamp":{
      "type":"Date",
      "default":"Date.now"
}
```

Listing 10: Posible esquema de auditoría

4.1.3. Notas sobre la implementación

En esta sección se dejan notas de las distintas decisiones que se tuvieron que tomar a la hora de la implementación.

Atributos que en la base de datos organizacional son un conjunto de elementos queda representada como un String en el metamodelo.

Al momento de realizar la carga del metamodelo desde la base de datos de Neo4j se identificó que los nodos y relaciones pueden compartir el identificador, esto genera un problema de claves duplicadas al momento de realizar la inserción en la base de datos del metamodelo. La forma de solucionar este problema fue tomar como identificador no solo al id del nodo/relación, sino que también se le agrega el nombre de la entidad a la que se corresponde esa instancia. De esta forma el mapeo al metamodelo es, dado un id de instancia más el nombre de la entidad le corresponde un id único en la base de datos del metamodelo.

Dado que cada motor de base de datos puede tener identificadores numéricos o alfanuméricos la decisión es que todos los identificadores obtenidos de las bases de datos organizacionales los tomamos como strings y al momento de realizar el pasaje a la base de datos organizacionales se utilizan diccionarios para almacenar la información de que id de entidad le correspondió en PostgreSQL a cada

entidad cargada.

La idea y propósito general de la API No SQL es la de proporcionar un mecanismo genérico que le permita a los usuarios, mediante el uso de esta interfaz, cargar los datos de las bases de datos organizacionales hacia el metamodelo. Como su nombre lo indica, la API está diseñada para que pueda soportar cualquier tipo de esquema de Base de datos, en especial las NoSQL.

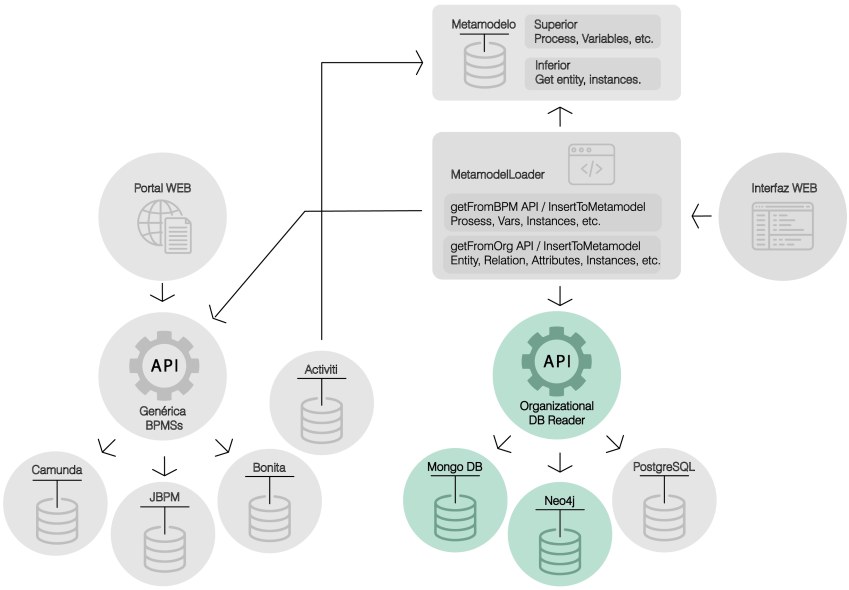

Figura 4.6: Organizational API

En la figura 4.6 se muestra donde se ubica este componente, dando un contexto general y el propósito de esta API. Uno de los ejemplos de utilización de esta API es el componente Organizational DB Reader.

Las operaciones que expone la API NoSQL se definen a continuación:

List<String>GetEntityNames()
Retorna una lista con los nombres de las entidades existentes.

List<Entity>GetEntityDefinitions()
Retorna la lista de entidades, incluyendo sus atributos.

Entity GetEntityDefinition(string entityName)
Retorna la información de la entidad dada por su entityName Si la entidad no existe se retorna NULL.

List<EntityInstance>GetEntityInstances(string entityName)

Retorna todas las instancias pertenecientes a una entidad dado por su nombre Si la entidad no existe se retorna NULL.

List<String>GetRelationNames()
Retorna una lista con los nombres de las relaciones existentes.

List<Relation>GetRelationDefinitions()
Retorna la lista de relaciones, incluyendo sus atributos.

Relation GetRelationDefinition(string relationName)
Retorna la información de la relación dada por su relationName Si la relación no existe se retorna NULL.

List<RelationInstance>GetRelationInstances(string relationName)
Retorna todas las instancias pertenecientes a una relación dada por su nombre Si la relación no existe se retorna NULL.

4.1.4. Adaptaciones al Metamodelo

Con motivo de la extensión para la carga de distintos tipos de bases de datos es que surge la necesidad de realizar modificaciones en el esquema del metamodelo, puntualmente en como se almacenan las relaciones. Inicialmente se contaban con estructuras y operaciones para cargar relaciones entre entidades llamadas EntityEntity, pero tenían el inconveniente que no modelaban cuando dichas relaciones tenían atributos. Por esto se realizo un rediseño del metamodelo incluyendo la tabla Relation y por consiguiente la tabla RelationInstance. Las operaciones que soportan esta nueva estructura son:

List<String>GetRelationNames()
Retorna una lista con los nombres de las relaciones existentes.

List<Relation>GetEntityDefinitions()
Retorna la lista de relaciones, incluyendo sus atributos.

Relation GetRelationDefinition(string relationName)
Retorna la información de la relación dada por su relationName.Si la relación no existe se retorna NULL.

List<RelationInstance>GetRelationInstances(string relationName)
Retorna todas las instancias pertenecientes a una relación dada por su nombre. Si la relación no existe se retorna NULL.

4.1.5. Proceso

La primera versión de la NoSQL API se comenzó a construir tomando como referencia la prueba de concepto. En esta instancia se planteó con sólo una operación teniendo la siguiente firma:

Data getDBEntities(Context context)

Como entrada para esta operación solamente se solicita la información llamada "contexto" con las credenciales de autenticación para la base de datos organizacional. Este método es instanciado por cada tipo de bases de datos mediante un cliente, el cual conoce la forma de representación de ese tipo de base de datos para obtener la información necesaria. Los datos obtenidos de la base de datos organizacional son entidades, relaciones, atributos e instancias de ellos. El valor de salida de este método contiene toda esa información. Al tener en un tipo de datos estructurado todo lo necesario de la base de datos organizacional solamente resta pasar esa información capturada al metamodelo.

Para una segunda versión de esta API se separa ese único método ofrecido en varios más pequeños brindando mayor granularidad en la recuperación de la información. La modificación más importante entre las distintas versiones es que todos los métodos ahora se exponen para ser consumidos dejando de estar encapsulado todo en una operación.

Se debe resolver el manejo de la conexión, en el caso de servicios REST mediante token de sesión, de manera que se requiera una única vez enviar los datos de conexión. En caso de que sea una API como dependencia se mantiene la referencia en memoria.

4.1.6. Prototipo de carga

Luego de la creación de la API, y previo a la implementación formal del componente que haga uso de esta API, se procede a realizar un prototipo que, dada una estructura interna que se lee desde los DBMS NoSQL organizacionales permite crear dichas instancias en la base de datos relacional del metamodelo. Este prototipo carga para cada entidad que identifica en la base de datos organizacional de origen sus atributos e instancias en el mencionado metamodelo, considerando la estructura del mismo como claves foráneas y tipos de datos definidos en él.

El algoritmo general consiste en ir insertando en la base las entidades sin dependencias y luego las que tienen sus dependencias (claves foráneas) ya insertadas. La secuencia es Entity, Relation, Attribute, EntityInstance, RelationInstance, AttributeInstance.

4.1.7. Algoritmo de carga

En la API hay definidas operaciones para distintos escenarios. El algoritmo más simple para la carga de las entidades y sus instancias en el metamodelo es

el que se muestra en el listado 11.

```
List<String> entitiesStr = getEntityNames()

foreach entityStr in entitiesStr
        Entity entity = getEntityDefinition(entityStr)

    dbMetamodelo.entity.insert(entity)
        foreach attribute in entity.Attributes
                dbMetamodelo.attribute.insert(attribute)
        endfor

        List<EntityInstance> instances =  GetEntityInstances(entity.Name)
        foreach instance in instances
        dbMetamodelo.entityInstance.insert(instance)
        foreach attributeInstance in instance.attributes
                dbMetamodelo.attributeInstance.insert(attributeInstance)
        endfor
    endfor
endfor

List<String> relationsStr = getRelationNames()

foreach relationStr in relationsStr
        Relation relation = getRelationDefinition(relationStr )

    dbMetamodelo.relation.insert(relation)
        foreach attribute in relation.Attributes
                dbMetamodelo.attribute.insert(attribute)
        endfor

        List<RelationInstance> instances =  GetRelationInstances(relation.Name)
        foreach instance in instances
        dbMetamodelo.relationInstance.insert(instance)
        foreach attributeInstance in instance.attributes
                dbMetamodelo.attributeInstance.insert(attributeInstance)
        endfor
    endfor
endfor
```

Listing 11: Implementación de algoritmo de carga al metamodelo inferior

4.2. API Genérica BPMS

La API genérica es una API REST que permite interactuar con distintos sistemas BPM. Este proyecto da soporte a tres plataformas distintas: Bonita, JBPM y Camunda. El propósito inicial de este trabajo no fue el de proveer operaciones específicas para que se carguen los datos necesarios en el metamodelo. Sin embargo, muchas de sus operaciones expuestas pueden ser reutilizadas con ese fin. Para que la API sea de utilidad completa para poder cargar el metamodelo, se deben definir nuevas operaciones a exponer, y también se deben ajustar operaciones existentes.

En la figura 4.7 se muestra el punto de partida de este proyecto.

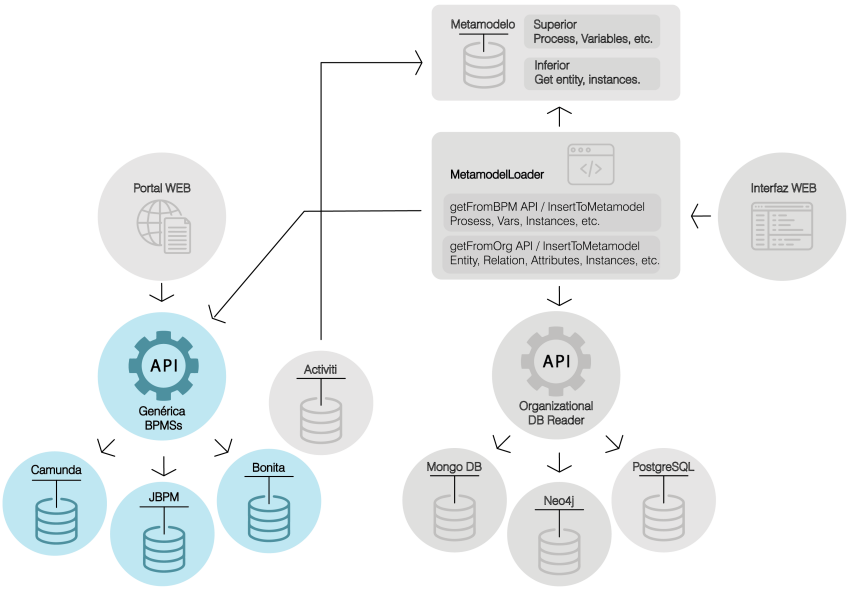

Figura 4.7: BPMS API

4.2.1. Extensión

Lo que se realiza es una adaptación de la API ya desarrollada con el objetivo de que la misma sea de utilidad a la hora de brindar la información necesaria para la carga de la base de datos del metamodelo, proveniente de los motores de procesos. Como primer punto se define el motor de proceso a utilizar para realizar las modificaciones de las operaciones y su implementación correspondiente. Elegimos Camunda ya que tiene una gran similitud con el motor Activiti, siendo un fork de éste [55]. La ventaja es que se puede tener una base para la carga de esta sección en el metamodelo ya que existe una implementación utilizando

Activiti. Luego de esto el objetivo es levantar en un ambiente local esa API genérica y probar sus operaciones. Luego del relevamiento de las operaciones disponibles es que se pasa a planificar el algoritmo de carga del metamodelo para los datos obtenidos en el motor de procesos.

A continuación se listan las operaciones mas importantes que son de utilidad para la obtención de los datos de los motores de procesos.

- getUsers
- getGroups
- getTasks
- getProcesses

A su vez, debido a las necesidades de este caso, se agregan implementaciones de operaciones para el motor Camunda, que inicialmente no se realizaron.

- *getGroups*: Retorna una lista de grupos con sus usuarios. Se implementa únicamente para el motor Camunda, dejando como trabajo futuro las implementaciones de los restantes motores.
- *getURLMotor*: Retorna el URL del motor de donde provienen los datos de procesos.

Por otro lado, se listan las operaciones a las que se le debieron realizar acondicionamientos y adaptaciones para este nuevo escenario de uso.

- *getTasks*: Se modifica la firma de la función, agregando dos nuevos parámetros. El motivo del cambio es que la llamada a la API en la implementación de Camunda se modifica, obteniendo el conjunto de tareas más adecuado a lo que se espera. Se utiliza el endpoint que provee Camunda para cargar el histórico de tareas (history/activity-instance). Con la llamada utilizada en versiones anteriores sin el histórico, solamente se obtenían tareas de procesos no finalizados. Esto era una limitante importante en los resultados obtenidos y por ese motivo se decide modificar la operación a utilizar de la API de Camunda. Con el cambio de endpoint a utilizar se modifica la firma, pasando dos nuevos parámetros que son el id de proceso, es decir, el proceso al que pertenece y su definición de tarea.
- *getCases*: Como se utiliza el histórico para obtener esta información(history/process-instance) se agrega también el identificador del proceso como parámetro para realizar la consulta a la API de Camunda, esto permite reducir la cantidad de resultados y poder procesar los cases de manera ordenada.
- Clase *Process*: Se agrega soporte para roles y usuarios asociados.
- Clase *TaskDefinition*: Se agrega soporte para roles y usuarios asociados.
- Clase *VariableInstance*: Las instancias de variables inicialmente no hacían uso de la fecha de creación de la misma. El metamodelo lo requiere, por lo que se agregó en la estructura.

4.2.2. Algoritmo BPMS

Luego de tener los ajustes para el nuevo uso de la API genérica es que nos encontramos en condiciones de definir el algoritmo de carga del metamodelo, donde las fuentes de datos son los motores de procesos BPMS. Al comienzo, para crear el algoritmo, fue necesario definir los órdenes de precedencia entre los distintos componentes. En el listado 12 se muestra la implementación del algoritmo de carga de la parte superior del metamodelo.

4.3. Metamodel Loader

El Metamodel Loader es un componente central de este proyecto [56]. Su función principal es interactuar con los proyectos que fueron definidos a partir de la sección 4. Se encarga de integrar la API que lee desde las bases de datos organizacionales y la API que lee desde los BPMS.

Con la incorporación de este componente se busca cumplir con uno de los objetivos de este proyecto que es es el desacoplamiento de la carga de la base de datos del metamodelo.

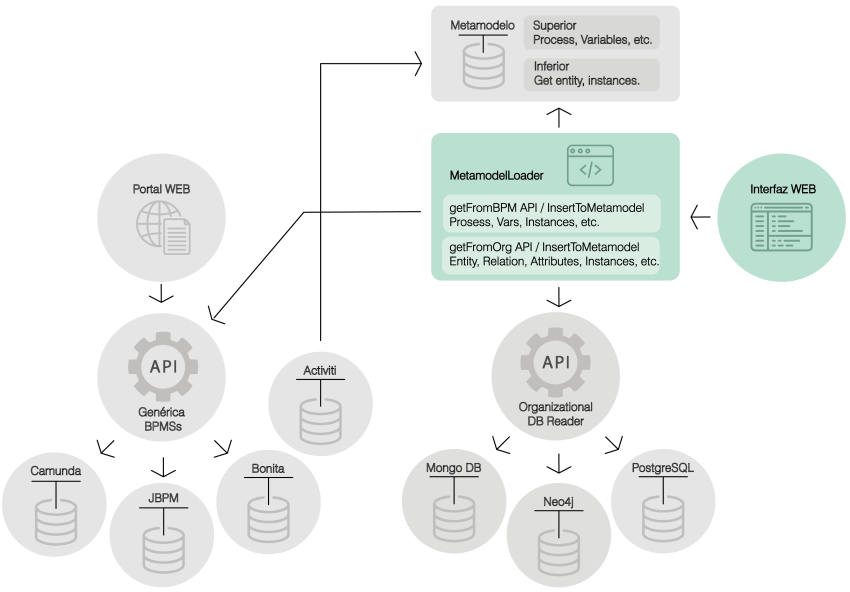

Figura 4.8: Metamodel Loader

En la figura 4.11 se presenta nuevamente un esquema general del proyecto. El punto a desarrollar en este capítulo comprende a los componentes resaltados en color verde.

4.3.1. Arquitectura

Es una API REST con dos componentes principales, uno para lectura de datos de motores BPMS y otro para lectura de bases de datos organizacionales.

Cada componente expone varios puntos de entrada para la lectura de la información y requieren los datos de conexión que requiera el sistema destino para conectarse. Un problema encontrado aquí es que la API genérica no recibe datos de conexión, sino que los lee de un archivo de configuración en el servidor de aplicaciones, lo cual no hace accesible la información del origen de los datos. Para esto se agrego una operación a la API genérica que permite obtener un identificador del motor al cual esta conectado.

Las operaciones expuestas por esta API son listadas a continuación:

read

Expone operaciones para leer información tanto de las bases de datos organizacionales tales como entidades, relaciones, instancias; como datos de procesos de BPMS. No carga en el metamodelo, sirve para explorar los datos.

load

Expone la operación *load(ConversionData data)* que recibe información del origen de los datos, ya sean BPMS o una BD organizacional, y del metamodelo destino, realizando la carga.

4.3.2. Interfaz Web

Para un uso mas cómodo de la API se provee una aplicación web básica que permite interactuar con ella.

En la figura 4.9 se ubica este componente dentro del esquema general del proyecto. La idea de este componente es simplificar el proceso de carga hacia el metamodelo desde las dos posibles fuentes de información que son desde procesos o desde bases de datos organizacionales. Una funcionalidad para extender el uso de este componente es que se permite al usuario consultar previamente a la carga masiva al metamodelo con el objetivo de validar inicialmente los datos. Por lo tanto la web se divide en 4 funcionalidades principales:

- Leer datos de una base de datos organizacional
- Leer datos de un motor BPMS
- Cargar datos de una base de datos organizacional al metamodelo
- Cargar datos de un motor BPMS al metamodelo

El la figura 4.10 se muestra la pantalla que permite realizar la carga completa de los datos organizacionales a la base de datos del metamodelo.

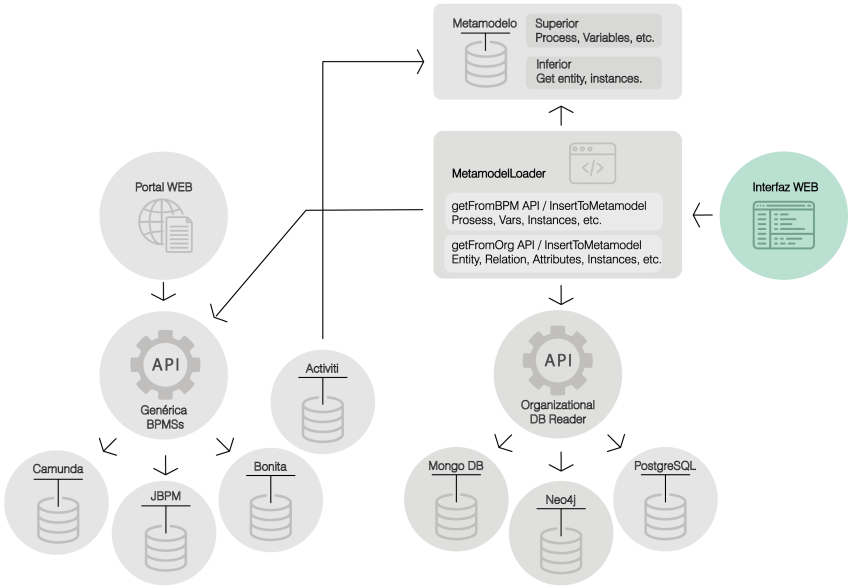

Figura 4.9: Metamodel Loader

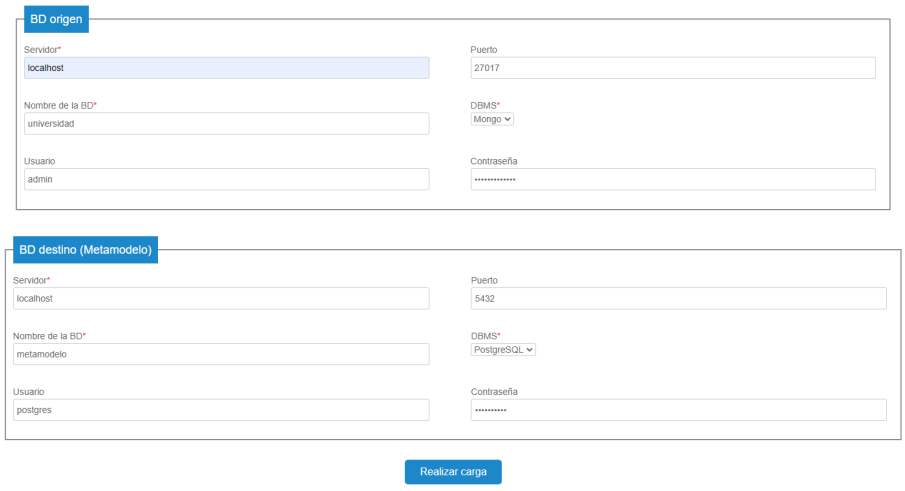

Figura 4.10: Pantalla de carga de datos organizacionales

4.3.3. Cambios al metamodelo

En el transcurso del proyecto se fueron modificando ciertas entidades y relaciones entre las mismas respecto al esquema original. En la figura 4.11 se muestra cómo queda el metamodelo luego de los cambios aplicados. A continuación se explican cada uno de ellos.

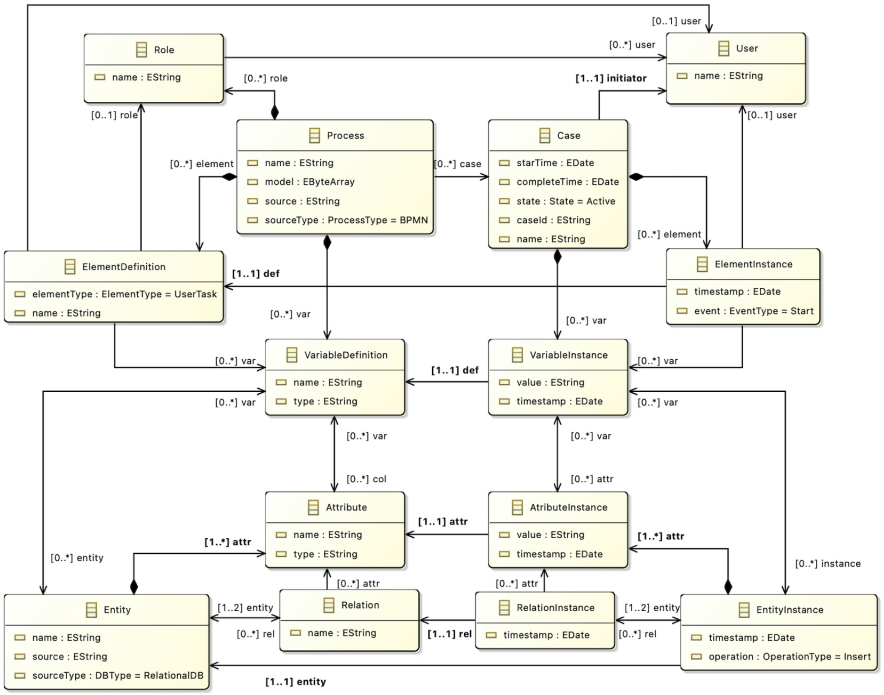

Figura 4.11: Metamodelo

Nuevas tablas

Tablas que fueron agregadas.

- relation: modela relaciones entre entidades
- relationinstance: instancias de relaciones entre entidades
- processuser: relación entre procesos y usuarios que pueden iniciarlos
- processrole: relación entre procesos y roles que pueden iniciarlos
- elementuser: relación entre tareas y usuarios que pueden tomarlas
- elementrole: relación entre tareas y roles que pueden tomarlos

Sourcetype:

- idsourcetype: int (PK)
- description: varchar {RelationalDB, DocumentDB, GraphDB, KeyValueDB, ColumnDB}

Processtype:

- idprocesstype: int (PK)
- description: varchar {BPMN, CMMN}

Nuevos campos

Entity:

- source: Un identificador de la fuente desde donde fue leída la entidad
- sourceType: para indicar la fuente de los datos. Por ejemplo Relacional, Documental, etc.

Attribute:

- idrelation: ya que un atributo ahora puede pertenecer a una relación
- identity ahora es nullable, ya que puede estar asociado a una relación
- nueva restricción para que identity e idrelation pueda ser solamente uno de ellos no null

Attributeinstance:

- idrelationinstance: int (FK)
- identityinstance ahora es nullable
- nueva restricción para que identityinstance e idrelationinstance pueda ser solamente uno de ellos no nulo

Campos modificados

Attributeinstance:

- se cambió el tipo del campo value a *text* para que acepte valores más grandes

Elementdefinition:

- se cambia el tipo de la clave primaria a integer

Tablas eliminadas

Se eliminan las tablas entityentity y entityinstanceentityinstance ya que no modelaban de manera completa la realidad. De cierta forma fueron renombradas a relation y relationinstance.

```
call API getGroups;
call API getTasks;
call API getProcesses;

for(User u in users)
    insertUser(u);
endfor

for(Group g in groups)
        insertRole(g);
        insertUserRole(g, g.getUsersGroup());
endfor

for(Process p in processes)
        insertProcess(p);
        if (p.getVariableDefinitions > 0)
        for(VariableDefinition vd in p.getVariableDefinitions())
                insertVariableDefinition(vd,p);
        endfor
        endif

        // carga definiciones de tareas y variables
    for(TaskDefinition td in p.getTaskDefinitions())
            insertElementDefinition();
        if (td.getTaskVariables() > 0)
            for (VariableDefinition vd in td.getVariableDefinitions())
                insertVariableDefinition(vd,td);
            endfor
        endif
    endfor

    // carga instancias de procesos y tareas
    cases = getCasesByProcessDefinition();
    for(Case c in cases)
        insertCase(c);
          // carga instancias de variables
        if(c.getCaseVariables() > 0)
            for(VariableInstance vi in c.getCaseVariables())
                insertVariableInstance(vi,c);
            endfor
        endif
    endfor

    // carga instancias de tareas
    for{(TaskDefinition td: taskDefinitions)}
        for(TaskInstance ti in td.getTaskInstances())
            insertElementInstance(ti);
        endfor
    endfor

endfor
```

Listing 12: Implementación de algoritmo de carga al metamodelo superior

Capítulo 5

Aplicación: Mobility

Con el objetivo de validar todo el trabajo realizado hasta el momento, desde el procesamiento de los datos de bases de datos no organizacionales, hasta el motor de procesos de Camunda se procede a aplicar todo esto en un caso de estudio.

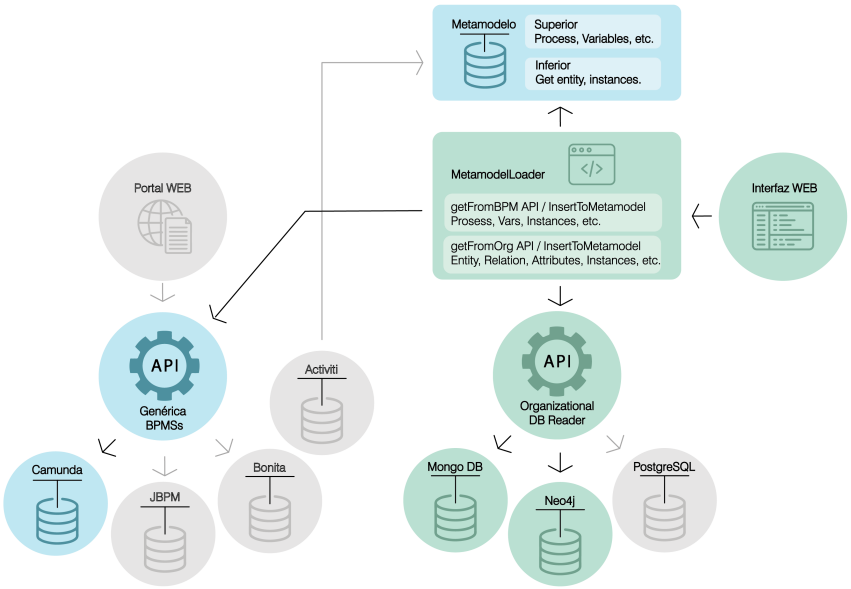

Figura 5.1: Componentes involucrados

En la figura 5.1 se destacan todos los componentes que participan en este ejemplo.

El caso de estudio es el asociado a la movilidad de estudiantes [6]. Este

ejemplo se puede modelar con un Business Process, modelándolo con la versión 2.0 de BPMN, como puede verse en la figura 5.2. Una movilidad se define como un programa donde los estudiantes pueden postularse a estudiar, revalidando materias en universidades extranjeras. Cada programa proporciona un monto estipulado para que los estudiantes puedan solventar sus estudios en el exterior.

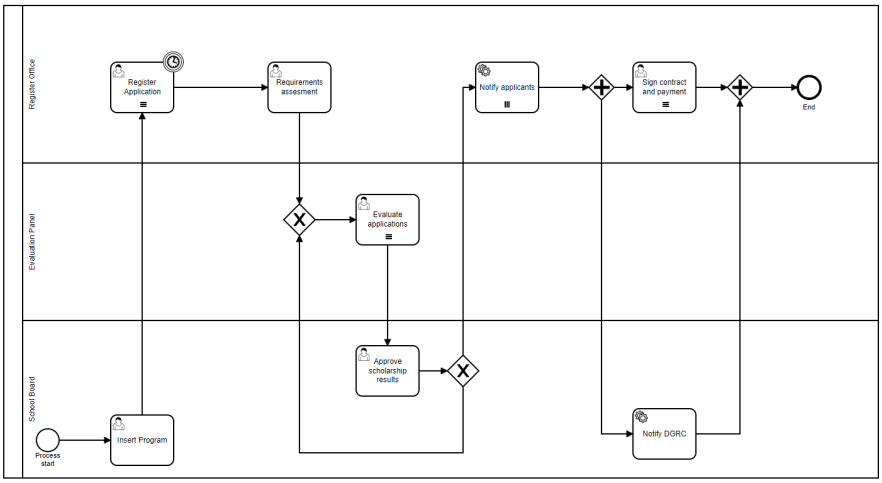

Figura 5.2: Proceso Mobility

Inicialmente los programas son evaluados por el consejo de la universidad que corresponda. Son éstos los que determinan los estudiantes que aprueban y cumplen cada programa. Los estudiantes pueden registrarse a un programa utilizando la oficina de registros siempre y cuando se encuentren dentro de los quince días del comienzo del mismo. Una vez finalizado el plazo de registro, la oficina de registro procesa todas las postulaciones, confirmando únicamente las que cumplen los requerimientos.

Las postulaciones que aprueben la primera etapa pasan al panel de evaluaciones. En esta etapa se evalúan y generan una lista de prioridades según méritos, quedando definidas las postulaciones prioritarias, es decir, titulares y también las que cuadran como suplentes. Todas las postulaciones categorizadas son enviadas al consejo que procederá con la aprobación. Si el consejo aprueba las listas, se procede a notificar a cada estudiante. Acto seguido, se notifica también a la DGRC, para finalizar con el pago correspondiente a los estudiantes, según se estipula en el programa.

El proceso de postulación, aprobación y adjudicación de un programa a los distintos estudiantes es un proceso de negocios que puede ser desarrollado e implementado por un BPMS. Para el caso de estudio la solución se desarrolla en Camunda. El modelo se compone de ocho tareas en total. Seis de ellas son tareas de usuario y las restantes son tareas de ejecución automática. Las tareas de usuario pueden ser tomadas por tres de los diferentes roles definidos. En la

figura 5.2 se puede observar la composición y distribución del modelo donde cada carril representa uno de los roles.

En cada uno de los pasos del proceso se emula el comportamiento de un proceso que ocurre en la realidad, registrando todos los datos correspondientes en una Base de Datos relacional del motor de procesos. Por otro lado, la información que almacena los datos correspondientes bases de datos organizacionales son persistidos en modelos no relacionales. Algunos datos organizacionales son leídos desde una base de datos documental utilizando MongoDB, mientras que el almacenamiento de los datos se realiza utilizando una base de datos orientada a grafos. Para este propósito se utiliza Neo4j. A continuación se presenta como se particionó la información organizacional, identificando qué manejador almacena cada entidad.

En MongoDB se almacenan las siguientes entidades:

- Cursos
- Docentes
- Carreras
- Institutos
- Estudiantes

Para almacenar los datos relacionados a las solicitudes se utiliza Neo4j:

- Solicitud
- Estado
- Programa
- Movilidad

Para mantener consistencia entre ambas bases de datos, el ejemplo debe realizar algunos controles básicos, es decir, para insertar una nueva solicitud en la base de datos de Neo4j el sistema debe controlar que previamente el estudiante que realiza la inscripción se encuentra en la base de datos de MongoDB.

Notar que los datos almacenados en MongoDB no son cargados como producto de la ejecución del proceso, sino que solamente son leídos con el objetivo de brindar información a los usuarios de Camunda que lo ejecutan y hacer validaciones de los datos ingresados por estos.

A continuación se describen todas las entidades involucradas en el proceso:

- Institute: institutos que participan de la organización.
- Teacher: Profesores que participan de la organización y pertenecen a un instituto.
- Course: Cursos pertenecientes a una carrera dentro de la organización.

- Career: Carreras de la organización.
- Student: Estudiantes que pertenecen a una carrera dentro de la organización.
- Program: Corresponde a todos los programas que fueron definidos en los procesos ejecutados.
- Application: Todas las solicitudes que se realizaron para los programas. Cada una de ellas está relacionada con un solo programa, un estudiante y se le asocia un estado.
- State: modela los diferentes estados que puede tener una solicitud.
- Mobility: Son todas las movilidades que fueron aprobadas con sus respectivos montos asignados, junto con la solicitud asociada.

Los identificadores de los nodos que se almacenan en Neo4j son numéricos generados automáticamente. El motor de base de datos asigna un identificador global para todos los nodos (instancias) que se van insertando. La decisión en este caso es aprovechar ese mecanismo y utilizar el identificador creado automáticamente por Neo4j para las instancias de entidades.

Ejecución de procesos

Se ejecutaron 10 instancias del proceso, con 3 solicitudes ingresadas por cada programa. Esto da un total de 30 solicitudes. La estructura de nodos puede verse en la figura 5.3. Los datos fueron cargados manualmente de forma de tener solicitudes en varios estados, generando todos los caminos posibles de ejecución dentro del proceso.

Por ejemplo para el Programa 10, de las 3 solicitudes ingresadas, una fue aprobada como titular y las restantes rechazadas por exceder el monto asignado al programa. Esto puede verse en la figura 5.4.

Carga del metamodelo Una vez que los procesos fueron disparados y finalizados en Camunda, se procede a la carga de los datos organizacionales y de procesos. Para esto se hace uso del componente MetamodelLoader presentado en el capitulo 8. En primera instancia se cargan los datos de la universidad desde MongoDB, luego los generados por el proceso en Neo4j y finalmente los datos de Camunda. El orden no es relevante ya que no hay dependencia entre los datos aun. Habrá dependencia siempre que el algoritmo de matching que se presenta a continuación sea capaz de relacionar las instancias cargadas.

5.0.1. Algoritmo de matching

El algoritmo de matching es un componente implementado en proyectos anteriores cuya función principal es operar sobre la base de datos del metamodelo

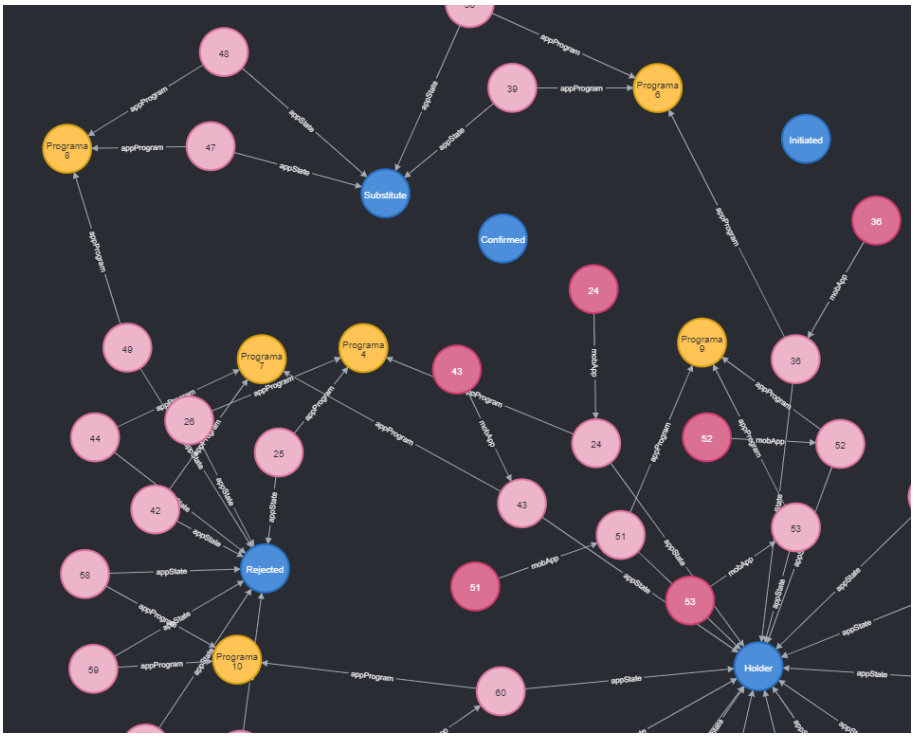

Figura 5.3: Vista de nodos en Neo4j

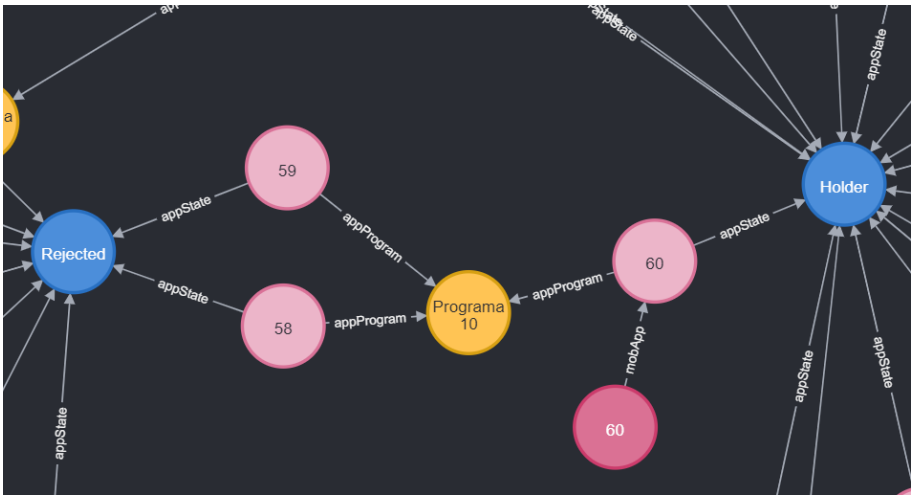

Figura 5.4: Ejemplo nodos generados

conectando entre los datos cargados del motor de procesos y los datos originados en la base de datos organizacional. Puntualmente lo que busca es unir las entidades AttributeInstance del lado de los datos organizacionales con sus respectivas VariableInstance del motor de procesos. A partir de ese match, se busca agregar las siguientes relaciones:

- AttributeInstance y VariableInstance
- Attribute y VariableDefinition
- Entity y VariableDefinition
- EntityInstance y VariableInstnce

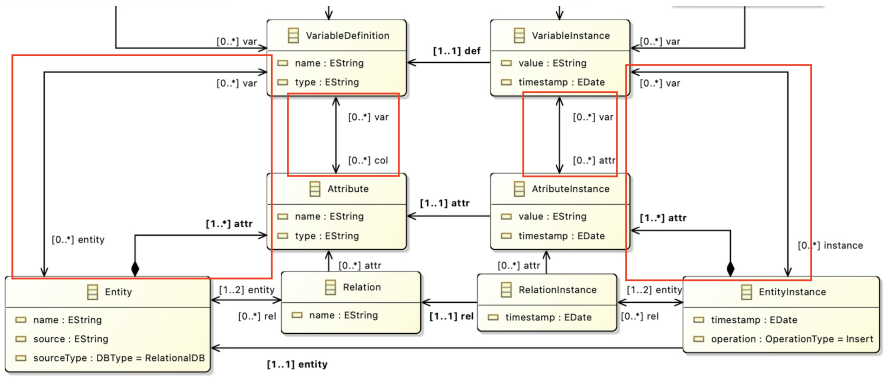

Figura 5.5: Metamodelo Matching

En la figura 5.5 se muestra en rojo todas las relaciones que apunta a resolver este algoritmo.

El algoritmo utiliza el timestamp como elemento clave al momento de realizar el matching, por lo que es de suma importancia tener registros de cuando se insertó cada dato, tanto en las bases de datos de procesos como las organizacionales. Para las bases de datos organizacionales NoSQL se utiliza la primera opción definida en la sección XX, es decir, se agrega un atributo timestamp extra en todos los nodos insertados desde. Este atributo será el utilizado al momento de cargar al metamodelo y hacer posible el matching entre los distintos datos.

5.0.2. Ejecución del algoritmo

- Utilizando la documentación proporcionada, el primer paso consiste en levantar el proyecto ETLMetamodel del proyecto pmdatos.
- Se realizan pequeñas modificaciones al código, teniendo en cuenta los cambios realizados en el metamodelo. Básicamente cambios de nombres y/o tipos de atributos de las tablas.

Cuadro 5.1: Atributos de la entidad Program

Atributo	Camunda	Matcheado?	Variable	Correcto?
date	si	no	-	-
idprogram	no	-	-	-
timestamp	no	-	-	-
amount	si	si	Total Amount	si
year	si	si	Year	si
name	si	si	Program Name	si
callnumber	si	si	Call Number	si

- Ejecución del algoritmo invocando el servicio .../match
- Análisis de los resultados obtenidos

5.0.3. Resultados obtenidos

El algoritmo de matching produjo los resultados que se presentan a continuación. Para darle contexto a los resultados es necesario conocer los datos sobre los cuales opera.

En el metamodelo se cuenta con 149 instancias de variables cargadas desde Camunda, producto de la ejecución de 10 procesos completos del flujo de Mobility.

En cuanto a instancias de atributos se cargaron un total de 477, de las cuales 405 provienen de Neo4j, la única base en la cual se almacenan datos ingresados desde el flujo en Camunda. Los datos de MongoDB son solamente leídos por los procesos.

El resultado arrojado por el algoritmo es de 54 emparejamientos entre instancias de variables y atributos. Este valor implica un 13 % de los atributos y un 36 % de las variables.

Sin embargo, haciendo un análisis de los atributos que no fue capaz de relacionar vemos que hay casos en que los valores no se desprenden de datos ingresados desde variables de proceso sino que son autogenerados (como los identificadores de programas y solicitudes) o salen de valores ya existentes (movilidades a partir de las solicitudes). Se presenta entonces desglosado por entidad un detalle de cada atributo.

Program

Esta es la entidad que presenta mayor porcentaje de emparejamientos exitosos. Ver el cuadro 5.1.

Aciertos: 4 de 5

Viendo el motivo de las fechas es por la representación de ambos, la forma de serializar a texto la fecha no coincide y luego se usa un comparador de igualdad de strings. Una mejora a realizarse entonces es que el algoritmo de

Cuadro 5.2: Atributos de la entidad Application

Atributo	**Camunda**	**Matcheado?**	**Variable**	**Correcto?**
idapplication	no	-	-	
date	no	-	-	
approved	no	-	-	
notified	no	-	-	
timestamp	no	-	-	
idstudent	si	si	Requested Amount	no
order	no	si	Student Id	no
amount	si	si	Requested Amount	si

Cuadro 5.3: Atributos de la entidad Mobility

Atributo	**Camunda**	**Matcheado?**	**Variable**	**Correcto?**
amount	si	no	-	-
date	no	-	-	-
startdate	si	no	-	-
idmobility	no	-	-	-
timestamp	no	-	-	-

matching tenga en cuenta el tipo del dato que está comparando para identificar si representan el mismo valor, de forma de no descartar como en este caso emparejamientos.

State

Es una entidad precargada, no es afectada por el proceso de Camunda.

Application

Las solicitudes tienen algunos datos que se ingresan directamente desde Camunda y luego otros campos como notified y approved sea actualizan en otras etapas del proceso. Ver el cuadro 5.2.

Aciertos: 1 de 3

En el caso de la confusión entre order y Student Id es porque los dominios coinciden en algunas instancias. Puntualmente en 6 de 30 instancias las asocia. Esto es en la tarea donde se asigna un orden a las solicitudes.

Mobility

Sus datos no provienen de variables creadas en el mismo momento que los datos son ingresados, por lo tanto el algoritmo de matching no es capaz de relacionar sus atributos con ninguna variable. Ver el cuadro 5.3.

Aciertos: 0 de 2

A nivel de definiciones entonces tenemos 5 de 10 de matcheos correctos.

Capítulo 6

Conclusiones y Trabajo Futuro

6.1. Conclusiones

Adaptarse a las nuevas corrientes tecnológicas y a diferentes formas de almacenar grandes volúmenes de información es un punto en donde las organizaciones hacen mucho foco. Alineados con esto, las investigaciones en curso que buscan mejorar el funcionamiento automatizado de las organizaciones, y generar una visión unificada entre los datos también deben seguir el mismo camino.

En este trabajo hemos explorado cómo se puede extender la extracción de la información de las bases de datos organizacionales extendiendo el soporte para bases de datos NoSQL utilizando una API. Para validar el funcionamiento de esta API creada implementamos las operaciones expuestas para los manejadores MongoDB y Neo4j, siendo éstos dos los manejadores más populares para bases de datos documentales y orientadas a grafos, respectivamente.

Por otro lado, realizamos adaptaciones en una API BPMS ya desarrollada [8] buscando que ésta pueda procesar la información necesaria proveniente de los BPMS hacia un metamodelo común exitosamente. La forma de validar todo el proceso en una visión unificada fue mediante un ejemplo práctico. Para este propósito utilizamos el BPMS Camunda. Con el objetivo que esta investigación sea de utilidad para el futuro y se alinee a investigaciones en curso finalizamos el ejemplo ejecutando el algoritmo de matching implementado en [9], completando ahora sí, una visión integrada de todo el proceso y todos los datos que pertenecen a una organización.

Los componentes desarrollados y modificados conforman un ecosistema que permite que la carga del metamodelo desde fuentes heterogéneas se realice de una manera estándar utilizando distintas APIs que pueden ser extendidas para cualquier otra tecnología que se quiera incorporar en un futuro.

Hemos ajustado y probado todo el proceso con el objetivo de que sea escalable y la inclusión de nuevos proyectos que enriquezcan el proceso sea más

sencillo, siendo esto último un aporte de gran valor para la investigación.

Como conclusión del proceso destacamos que la implementación de la API NoSQL tiene una dificultad dispar dependiendo del manejador de base de datos. Esto quedó evidenciado en las dos implementaciones realizadas. En Neo4j, la obtención de la información en la base de datos organizacional fue directa, en especial la identificación de las relaciones debido a que las bases de datos orientadas a grafos reconocen y representan a las relaciones como componentes concretos en la estructura de almacenamiento. En cambio, para MongoDB la dificultad fue superior ya que por construcción de las bases de datos NoSQL, la flexibilidad en la definición de esquema complicó la identificación de las relaciones y fue uno de los problemas que más trabajo dio resolver.

6.2. Trabajo futuro

A modo de cierre del proyecto, es de interés agregar las posibles mejoras a futuro que creemos pueden resultar como mejora en el producto final.

La API BPMS fue modificada solo en su implementación para Camunda. Las demás implementaciones existentes no se actualizaron para proveer todos los datos que requiere el MetamodelLoader. Un camino de mejora puede ser en este sentido, soportar más BPMSs.

En la lectura de datos de Camunda, se dio soporte para los formularios embebidos en la definición del XML del proceso. Hay otras variantes que no están soportadas en esta versión y podrían incluirse en futuras versiones.

Para poder relacionar mediante el algoritmo de matching datos de los procesos con datos organizacionales, se modificó la implementación del proceso de negocio de forma que el timestamp de inserción de los datos sea almacenado en la base organizacional. Conjuntamente en el MetamodelLoader se agregó una lógica que, en caso de las instancias no contar con timestamp definido, asigne el valor del atributo timestamp siempre que exista. La conversión de texto a fecha se hace utilizando el formato de hora local. Esto se hizo como alternativa a la idea inicial de contar con registros de auditorías de los DBMSs. La obtención de los registros de auditorías en la base de datos es un punto importante en lo que refiere a este proyecto. Se aplicaron las auditorías mediante código, pero a futuro es de interés aplicar la obtención de los logs sin afectar los registros de la base de datos organizacional. Como fue expresado a lo largo de esta investigación, existe dificultad al momento de obtener registros de auditoría. Una forma alternativa es modificar el algoritmo de matching de forma tal que se evite depender de los registros de auditoría al momento de conectar los datos.

Para la identificación de relaciones en Mongo DB implementamos un mecanismo para obtener un esquema a partir de los documentos JSON de las colecciones de la base de datos. Este componente al resolver ese problema es-

pecífico de interpretación de archivos JSON es potencialmente aplicable en otros motores de base de datos que almacenen los datos en ese formato, o incluso aplicable en otras áreas que sea de interés trabajar o interpretar datos a partir de un archivo JSON.

Finalmente, es interesante ver si la inclusión de otras familias de bases de datos NoSQL tiene algún impacto en la definición del metamodelo, dado que al considerar las bases de datos basadas en grafos resultó en la agregación de un nuevo componente. Por otro lado, si bien en este proyecto se hace una implementación distinta para cada DBMS, es importante estar atento al surgimiento de algún lenguaje de consulta común a varios motores que permita abstraer el acceso a alguna familia particular por ejemplo, así como SQL para las bases relacionales.

Referencias

[1] Weske, M. (2007). Business process management architectures (pp. 305-343). Springer Berlin Heidelberg.

[2] ¿Qué es un Data Scientist? (2015, May 20). InLab FIB. https://inlab.fib.upc.edu/es/blog/que-es-un-data-scientist

[3] White, S. A., & Miers, D. (2008). BPMN. Modeling and Reference Guide. Understanding and using BPMN. Develop rigorous yet understandable graphical representations of business processes. Future Strategies Inc., Lighthouse Point, Fla.

[4] SANS Institute: Information security resources. https://www.sans.org/information-security (Accedida el 16 de Diciembre de 2022)

[5] Elmasri, R., Navathe, S. B., Castillo, V. C., Pérez, G. Z., & Espiga, B. G. (2007). Fundamentos de sistemas de bases de datos (No. QA76. 9D3 E553 2007.). Pearson educación.

[6] Delgado, A., & Calegari, D. (2020, October). Towards a unified vision of business process and organizational data. In 2020 XLVI Latin American Computing Conference (CLEI) (pp. 108-117). IEEE.

[7] ¿Qué es el sistema manejador de bases de datos? Retrieved December 16, 2022, from https://blog.powerdata.es/el-valor-de-la-gestion-de-datos/bid/406549/qu-es-el-sistema-manejador-de-bases-de-datos

[8] Belén Remedi, Diego Rodriguez y Alejandro Guggeri. Espacio GitLab. Disponible en: https://gitlab.fing.edu.uy/opencoal/portalbpms. (Accedida en Agosto de 2021)

[9] Andrés Borges, Alexis Artus. Espacio GitLab. Disponible en: https://gitlab.fing.edu.uy/COAL/pmdatos (Accedida en Noviembre de 2022)

[10] Sitio Oficial Forrester, https://www.forrester.com/

[11] Business process. TechTarget, https://www.techtarget.com/searchcio/definition/business-process (Accedida el 21 de Diciembre de 2022)

[12] Sitio oficial BPMN, https://www.bpmn.org/

[13] Sitio oficial OMG, https://www.omg.org/

[14] BPMN 2.0 Poster http://www.bpmb.de/images/BPMN2_0_Poster_EN.pdf (Accedida en Noviembre de 2022)

[15] jBPM Business Automation Toolkit. (n.d.). JBPM. Accedida el 16 de Diciembre de 2022, de https://www.jbpm.org/

[16] Bonita, sitio oficial, https://es.bonitasoft.com/ (Accedida en Noviembre de 2022)

[17] Camunda, sitio oficial, https://camunda.com/ (Accedida en Noviembre de 2022)

[18] Activiti, sitio oficial, https://www.activiti.org/ (Accedida en Noviembre de 2022)

[19] Documentación API REST de Camunda, https://docs.camunda.org/manual/7.16/reference/rest/ (Accedida en Noviembre de 2021)

[20] https://camunda.com/blog/2022/02/moving-from-embedded-to-remote-workflow-engines/ (Accedida en Noviembre de 2022)

[21] Jing Han, Haihong E, Guan Le, Jian Du, Survey on nosql database, in: 2011 6th International Conference on Pervasive Computing and Applications (ICPCA), 2011

[22] https://developer.mozilla.org/es/docs/Glossary/CRUD (Accedida en Noviembre de 2022)

[23] Gudivada, V.N., Rao, D., Raghavan, V.V., 2014. NoSQL Systems for Big Data Management, IEEE World Congress on Services 2014 http://dx.doi.org/10.1109/ SERVICES.2014.42, pp. 190e197, June 27 2014eJuly 2 2014.

[24] Furht, B. and Villanustre, F. (2016). Introduction to big data. In Furht, B. and Villanustre, F., editors, Big Data Technologies and Applications, pages 3–11. Springer

[25] Cálculo de rankings del sitio DB Engines https://db-engines.com/en/ranking_definition (Accedida en Abril de 2021)

[26] Ranking de bases de datos documentales, DB Engines https://db-engines.com/en/ranking/document+store (Accedida en Abril de 2021)

[27] Ranking de bases de datos basadas en grafos, DB Engines https://db-engines.com/en/ranking/graph+dbms (Accedida en Abril de 2021)

[28] The Forrester Wave™: Graph Data Platforms, Q4 2020. Reporte.

[29] The Forrester Wave™: Big Data NoSQL, Q1 2019. Reporte.

[30] Referencias entre documentos, MongoDB https://docs.mongodb.com/manual/reference/database-references/ (Accedida el 14 de Abril de 2021)

[31] Mongo DB, sitio oficial, https://mongodb.com

[32] Amazon DynamoDB API https://docs.aws.amazon.com/amazondynamodb/latest/APIReference/Welcome.html (Accedida el 14 de Abril de 2021)

[33] DescribeTable Amazon DynamoDB API https://docs.aws.amazon.com/amazondynamodb/latest/APIReference/API_DescribeTable.html (Accedida el 14 de Abril de 2021)

[34] INFER. Couchbase Docs. https://docs.couchbase.com/server/current/n1ql/n1ql-language-reference/infer.html (Accedida el 14 de Abril de 2021)

[35] Cliente Java para Couchbase https://docs.couchbase.com/sdk-api/couchbase-java-client/ (Accedida el 14 de Abril de 2021)

[36] Grimaldi, R.P., Matemáticas discreta y Combinatoria, Una Introducción con Aplicaciones 3ed

[37] What is OLTP? https://www.ibm.com/cloud/learn/oltp (Accedida el 14 de Abril de 2021)

[38] https://neo4j.com/graphacademy/training-intro-40/01-neo4j-graph-database/ (Accedida el 14 de Abril de 2021)

[39] G. Harrison, Next Generation Databases, NoSQL, NewSQL, and Big Data, Apress, 2015.

[40] Angles, Renzo. (2012). A Comparison of Current Graph Database Models. Proceedings - 2012 IEEE 28th International Conference on Data Engineering Workshops, ICDEW 2012. 171-177. 10.1109/ICDEW.2012.31.

[41] BD Neo4j, https://neo4j.com/product/neo4j-graph-database/ (Accedida el 14 de Abril de 2021)

[42] Plugin APOC para Neo4j, https://neo4j.com/labs/apoc/4.1/installation/ (Accedida el 14 de Abril de 2021)

[43] Introducción a Cosmos DB, https://docs.microsoft.com/es-es/azure/cosmos-db/graph-introduction (Accedida el 14 de Abril de 2021)

[44] Cosmos DB, NoSQL, https://docs.microsoft.com/es-es/azure/cosmos-db/relational-nosql (Accedida el 14 de Abril de 2021)

[45] ArangoDB, sitio oficial, https://www.arangodb.com/ (Accedida el 14 de Abril de 2021)

[46] API de ArangoDB, https://www.arangodb.com/docs/stable/http/ (Accedida el 14 de Abril de 2021)

[47] GraphQL https://neo4j.com/press-releases/query-language-graph-databases-international-standard/ (Accedida el 14 de Abril de 2021)

[48] Unicidad de atributos en MongoDB, https://docs.mongodb.com/upcoming/core/index-unique/ (Accedida el 30 de Octubre de 2021)

[49] Mongo Schemas, https://docs.mongodb.com/realm/mongodb/document-schemas/ (Accedida el 19 de Octubre de 2021)

[50] Mongo Schema Validation, https://docs.mongodb.com/manual/core/schema-validation/ (Accedida el 19 de Octubre de 2021)

[51] Neo4j Graph Schema, https://neo4j.com/docs/getting-started/current/graphdb-concepts/#graphdb-schema (Accedida el 19 de Octubre de 2021)

[52] Schemas en BD de grafos, https://medium.com/@experoinc/schema-support-in-three-property-graph-databases-1beff569855e (Accedida el 19 de Octubre de 2021)

[53] Repositorio versión 1 de API NoSQL implementada para prueba de concepto https://gitlab.fing.edu.uy/german.gonzalez/nosqlapi/-/tags/v1.0

[54] Auditoría de operaciones en MongoDB, https://docs.mongodb.com/manual/core/auditing/#audit-events-and-filter (Accedida el 19 de Octubre de 2021)

[55] Camunda fork de Activity, https://camunda.com/blog/2013/03/camunda-forks-activiti-and-launches (Accedida el 29 de Setiembre de 2022)

[56] Repositorio de código del componente MetamodelLoader en GitLab de la Facultad de Ingeniería, https://gitlab.fing.edu.uy/german.gonzalez/metamodel-loader