UNIVERSIDAD DE LA REPÚBLICA URUGUAY

FACULTAD DE INGENIERÍA

Herramienta para la Medición de Calidad de Datos en Fuentes Heterogéneas

Informe de Proyecto de Grado presentado por

Ramiro Bentancor, Julieta Dubra, Agustín Rodríguez

en cumplimiento parcial de los requerimientos para la graduación de la carrera de Ingeniería en Computación de Facultad de Ingeniería de la Universidad de la República

Supervisores

Laura González
Adriana Marotta

Montevideo, 3 de julio de 2025

Herramienta para la Medición de Calidad de Datos en Fuentes Heterogéneas por Ramiro Bentancor, Julieta Dubra, Agustín Rodríguez tiene licencia CC Atribución 4.0.

Agradecimientos

El desarrollo de este proyecto de grado representó el desafío final de nuestra formación en la carrera de Ingeniería en Computación de la Facultad de Ingeniería de la Universidad de la República, en Montevideo, Uruguay. Constituyó la culminación de muchos años de esfuerzo, durante los cuales recibimos el apoyo no solo de un cuerpo docente calificado y comprometido, sino también de nuestras familias, amistades y compañeros/as de carrera.

En particular, queremos expresar nuestro profundo agradecimiento a nuestras tutoras, Laura González y Adriana Marotta, por habernos acompañado a lo largo de este proceso. Valoramos enormemente sus aportes, su guía constante y los espacios que nos brindaron, tanto para asistir a presentaciones relacionadas con nuestra temática como para recibir retroalimentación que enriqueció nuestro trabajo.

También agradecemos especialmente a Silvana Pidre, por facilitarnos el acceso al modelo de calidad que desarrolló para el sistema nacional de salud, el cual sirvió como base para el caso de estudio de nuestro proyecto. Contar con un caso real nos permitió ampliar las funcionalidades del sistema, enfocarnos en mejorar la experiencia de usuario y perfeccionar detalles de implementación, lo que tuvo un impacto notable en el resultado final.

Asimismo, agradecemos al equipo de Salud.uy por su disposición a recibir una demostración del proyecto y por su interés en su mejora y desarrollo continuo.

Finalmente, queremos reconocernos mutuamente, como integrantes del equipo, por la paciencia, la disposición y el vínculo de compañerismo que supimos construir, no solo durante este proyecto, sino a lo largo de toda la carrera.

Resumen

La calidad de los datos se ha vuelto un factor fundamental para la toma de decisiones y el funcionamiento eficaz de las organizaciones. La calidad de datos puede definirse como el grado en que los datos satisfacen los requisitos establecidos para su uso, y su gestión requiere de un proceso estructurado que permita medir, evaluar y mejorar dicha calidad de forma continua. En este contexto surge este proyecto como una continuación de trabajos anteriores centrados en brindar soporte a dicho proceso de calidad de datos dentro de un marco conceptual formalizado.

Una iniciativa anterior desarrolló una plataforma orientada a facilitar la aplicación de un marco de referencia para la gestión de la calidad de datos en organizaciones. Dicha plataforma proporcionaba herramientas para caracterizar escenarios, analizar las propiedades de los datos y definir modelos de calidad adaptados a contextos específicos. Sin embargo, no abordaba de forma completa la etapa de medición y evaluación de la calidad de los datos, ni ofrecía soporte adecuado para trabajar con modelos y fuentes de datos heterogéneos.

Como respuesta a estas limitaciones, se desarrolló una nueva solución enfocada específicamente en fortalecer el soporte para la etapa de medición y evaluación de la calidad de datos. La aplicación implementada permite definir modelos de calidad, asociarles métodos de medición específicos, y ejecutar evaluaciones tanto de forma manual como programada. Además, incorpora soporte para múltiples tipos de fuentes de datos, incluyendo bases de datos relacionales, bases de datos documentales y archivos. La arquitectura de la solución propuesta fue diseñada para ser modular y extensible permitiendo, por ejemplo, agregar o remplazar componentes para ejecutar métodos sin necesidad de modificar el núcleo del sistema.

La solución fue evaluada mediante pruebas funcionales, validaciones con usuarios y análisis de escenarios de uso, permitiendo comprobar mejoras en la cobertura del proceso de calidad de datos y en la adaptabilidad de la herramienta a distintos entornos organizacionales.

Palabras clave: Calidad de datos, Proceso de gestión de la calidad de datos, Métodos de medición, Medición, Modelo de calidad, Fuentes de datos heterogéneas, Arquitectura modular

Índice general

1. Introducción **1**
- 1.1. Contexto y motivación . . . 1
- 1.2. Objetivos . . . 2
- 1.3. Aportes y resultados del proyecto . . . 3
- 1.4. Organización del documento . . . 3

2. Marco Teórico **5**
- 2.1. Calidad de Datos . . . 5
- 2.2. Marco para la Gestión de Calidad de Datos . . . 7
  - 2.2.1. Proceso de Gestión de Calidad de Datos . . . 7
  - 2.2.2. Modelo de Calidad de Datos de Referencia . . . 9
- 2.3. Arquitecturas de Big Data . . . 10

3. Revisión de Antecedentes **13**
- 3.1. Proyecto Fiumarelli y Fornio . . . 13
- 3.2. Herramientas Existentes . . . 15
  - 3.2.1. Aplicaciones . . . 15
  - 3.2.2. Librerías y Formatos . . . 17

4. Etapa de Análisis **21**
- 4.1. Roles dentro de la aplicación . . . 21
- 4.2. Requerimientos funcionales . . . 22
- 4.3. Requerimientos no funcionales . . . 24

5. Solución Propuesta **25**
- 5.1. Descripción General . . . 25
- 5.2. Principales Componentes . . . 27
  - 5.2.1. Portal Web . . . 27
  - 5.2.2. Lógica de Negocio . . . 27
  - 5.2.3. Procesamiento de Datos . . . 28
  - 5.2.4. Bases de Datos internas . . . 28
- 5.3. Soporte a Fuentes Heterogéneas . . . 29
- 5.4. Métodos Personalizados . . . 29
- 5.5. Agenda de *Jobs* y notificaciones . . . 31
- 5.6. Creación de Modelos . . . 32

5.7. Roles y usuarios 32

6. Implementación y Verificación **35**
6.1. Tecnologías utilizadas 35
6.2. Estructura de las bases de datos del sistema 36
6.3. Extracción de esquema y acceso a fuentes heterogéneas 42
6.4. Verificación 44

7. Caso de Estudio y Validación **47**
7.1. Presentación del diseño inicial de la arquitectura 47
7.2. Caso de estudio 48
7.3. Demo al equipo de Salud UY 60

8. Conclusiones y Trabajo Futuro **61**

Referencias **65**

Anexo A: Instructivo de creación de métodos personalizados **67**
A.1. Introducción teórica al sistema 67
A.1.1. Modelos de calidad de datos 67
A.1.2. Esquemas de Conexiones 68
A.2. Advertencia 72
A.3. Creación de métodos 72
A.3.1. Funcionalidades compartidas 72
A.3.2. Python 74
A.3.3. SQL 78
A.3.4. Validación de resultados 82
A.3.5. Evaluación de resultados 82

Anexo B: Formatos de JSONs importables para cargar modelos generales **83**

Anexo C: Historias de usuarios de la aplicación **85**

Capítulo 1

Introducción

Este trabajo surge como una continuación de un proyecto de grado anterior, combinado con la necesidad de automatizar ciertos aspectos del proceso de calidad de datos definido en el Marco de trabajo para la Gestión de la Calidad de Datos en Gobierno Digital (Computación y Agrimensura, 2020) creado por la Agencia de Gobierno Electrónico y Sociedad de la Información y del Conocimiento (AGESIC)[1]. En este capítulo se describe el contexto y la motivación del proyecto Herramienta para la Medición de Calidad de Datos en Fuentes Heterogéneas. Además, se presentan los objetivos del proyecto y los principales resultados obtenidos.

1.1. Contexto y motivación

La gestión de la calidad de los datos en las organizaciones implica procesos complejos compuestos por múltiples etapas. Estos procesos varían entre organizaciones, ya que los criterios de calidad que valoran pueden diferir según las prioridades y objetivos de cada una. En particular, en el caso de las organizaciones de gobierno, existen desafíos asociados al manejo de información de la población y al cumplimiento de normativas, por lo que resulta fundamental contar con mecanismos que asistan a este proceso.

En esta línea, la AGESIC, con el objetivo de sistematizar algunas actividades necesarias en la gestión de la calidad de datos a nivel gubernamental, y facilitar la comprensión de los elementos que deben incluirse, ofrece un marco de trabajo (Computación y Agrimensura, 2020) de ahora en adelante denominado "marco de referencia". El mismo incluye, entre otras cosas, un proceso para la gestión de la calidad de datos, que de ahora en más se denominará "proceso de calidad de datos", y un modelo de calidad de datos de referencia.

Para asistir en la aplicación de este marco, un proyecto de grado anterior (Fiumarelli y Fornio, 2020) desarrolló una herramienta que permite implementar algunas de las etapas del proceso de calidad de datos previamente mencionado. En particular

[1] https://www.gub.uy/agencia-gobierno-electronico-sociedad-informacion-conocimiento

aquellas relacionadas con la definición de la estrategia y modelo de calidad de datos, con la determinación de las causas de problemas de calidad y la definición de un plan de mejoras.

Sin embargo, el proyecto no abordó en profundidad la etapa denominada "Medir y Evaluar la Calidad de Datos". Esta fase, fundamental para obtener diagnósticos precisos sobre el estado de los datos, carecía de automatizaciones y soporte específico dentro de la herramienta. Por otro lado, la herramienta implementada no brindaba soporte a modelos y fuentes de datos heterogéneos. Este tipo de heterogeneidad en las fuentes de datos es, hoy en día, una característica clave y una necesidad en las organizaciones.

1.2. Objetivos

Dado el contexto, el objetivo de esta tesis es diseñar e implementar una herramienta que brinde soporte a la etapa de medición y evaluación de la calidad de datos, complementando y extendiendo las funcionalidades del proyecto anterior, con especial énfasis en el soporte a modelos de datos heterogéneos.

Para lograr esto, se plantean los siguientes objetivos específicos:

- **Investigar soluciones existentes para la gestión de la calidad de datos.** Dado que ya existen herramientas que asisten en este proceso, se apunta a explorar la posibilidad de conectarse a ellas para usarlas como base de esta nueva aplicación o recopilar información sobre sus arquitecturas y funcionalidades.
- **Analizar la problemática planteada.** Se apunta a estudiar el proceso de gestión de calidad de datos del marco de referencia, con el fin de entender las necesidades de los usuarios que participan en el proceso y definir requerimientos que se adecúen a ellas. Complementariamente, se proyecta estudiar la problemática del manejo de modelos de datos heterogéneos.
- **Diseñar una aplicación modular.** Este objetivo proyecta el diseño de la arquitectura de una herramienta con módulos que tienen objetivos concisos y son fácilmente reemplazables. Para mantener un bajo acoplamiento entre los componentes del sistema, se buscará además estandarizar su comunicación mediante un protocolo web.
- **Implementar un prototipo funcional.** Implementar una primera versión de la herramienta, que cumpla con la mayoría de las funcionalidades definidas. Se considerará sin embargo como un prototipo, ya que el alcance limita la cantidad de funcionalidades a implementar.
- **Verificar y validar el sistema** Este objetivo busca evaluar la aplicación definida en el punto anterior mediante pruebas de integración, revisiones de código, pruebas manuales, la implementación de un caso de uso real para probar la viabilidad del sistema y una presentación del mismo ante un grupo de interesados.

1.3. Aportes y resultados del proyecto

Teniendo en cuenta los objetivos definidos, los resultados y aportes del proyecto fueron los siguientes:

- **Relevamiento e investigación de herramientas disponibles en el mercado**: Se realiza un relevamiento de aplicaciones o herramientas que faciliten la implementación de la aplicación deseada. Se usan también como inspiración para el diseño de una nueva arquitectura.
- **Diseño de una arquitectura modular**: Se crea una arquitectura dividida en módulos, independientes entre ellos, que se comunican mediante interfaces que siguen el modelo de transferencia de estado representacional (REST). El diseño apunta a favorecer la re-usabilidad de los componentes del sistema y de los métodos usados para realizar mediciones en los datos.
- **Prototipo de la herramienta**: Se implementa un prototipo vertical de la herramienta objetivo, utilizando los lenguajes Python y Javascript, desplegándose la misma en un servidor de ElasticCloud. Este prototipo fue utilizado para validar las funcionalidades ofrecidas aplicando un modelo de calidad real con datos simulados.
- **Verificación y validación**: Se implementan pruebas automatizadas para verificar el sistema, creando casos de prueba regresivos con el fin de simplificar la incorporación de nuevas funcionalidades sobre lo ya desarrollado. Además se realizan presentaciones cuyas devoluciones dieron lugar a mejoras posteriores.

Mientras que el origen de este proyecto es inspirado en la problemática planteada por los casos del gobierno digital, la herramienta implementada fue diseñada para ser lo más versátil posible e ir más allá de este caso. Se llega entonces, a una herramienta que permite la creación de modelos de calidad variados, adaptados a cada organización y que da soporte a la creación de métodos re-utilizables para distintos tipos de fuentes de datos.

1.4. Organización del documento

El resto del documento se organiza de la manera detallada a continuación.

El Capítulo 2 presenta el marco teórico en el que se apoya el proyecto, y una serie de definiciones usadas a lo largo de todo el informe.

El Capítulo 3 presenta y analiza herramientas ya existentes, que se vinculan con el proyecto actual. Esto incluye la herramienta desarrollada por Fiumarelli y Fornio (Fiumarelli y Fornio, 2020), que se extiende en este proyecto, y además aplicaciones del mercado con funcionalidades similares a lo que se quiere crear. Se analizan adicionalmente formatos y herramientas que puedan ser usadas para facilitar la implementación del prototipo.

El Capítulo 4 aborda el revelamiento y análisis de requerimientos, funcionales y no funcionales, además de los roles necesarios dentro de la aplicación para abarcar los roles existentes dentro del proceso de calidad de datos.

El Capítulo 5 describe el diseño de la aplicación y los distintos componentes de la misma. Contiene descripciones del modelado de distintos elementos relevantes a la aplicación, como el modelo de calidad de datos y las fuentes de datos. Adicionalmente, se explican decisiones de diseño, como por ejemplo el manejo de diferentes tipos de fuentes de datos.

El Capítulo 6 abarca la implementación del prototipo, explica las decisiones tomadas para la implementación, y las herramientas utilizadas. Cuenta también con una sección que describe cómo se verificó el sistema.

El Capítulo 7 presenta la validación del sistema, y el caso de uso realizado para probar la aplicación. Sirve además como una guía de uso que presenta un flujo común de uso para la aplicación.

El Capítulo 8 corresponde a las conclusiones y aprendizajes obtenidos a partir del desarrollo del proyecto. Se analiza el trabajo realizado para evaluar el cumplimiento de los objetivos, e identificar posibles mejoras sobre el resultado.

Finalmente, el documento cuenta con tres anexos:

- Los formatos JSON que acepta la aplicación para importar modelos de calidad de datos al sistema.
- El documento completo de historias de usuario realizadas para el sistema.
- Un manual de usuario para la implementación de métodos dentro de la aplicación. Cuenta además con una serie de ejemplos de uso.

Capítulo 2

Marco Teórico

En este capítulo se presenta un marco teórico asociado a la calidad de datos, al marco de referencia para la gestión de calidad de datos de AGESIC, y a arquitecturas de big data.

2.1. Calidad de Datos

El término **calidad de datos** "se utiliza con referencia a un conjunto de características que deben poseer los datos, tales como su correctitud y su grado de actualización, entre otros" (Scannapieco y Catarci, 2002). También se define como "la capacidad de cumplir con los requerimientos definidos para el uso de los datos" (Olson, 2003). Esto es sumamente importante para las organizaciones que operan bajo la base de que los datos utilizados son válidos.

Con el objetivo de asegurar la calidad de sus datos, es que las organizaciones asumen la tarea de gestionarla. Se define entonces **la gestión de calidad de datos** como el trabajo de "medir, analizar, mejorar, y controlar, entre otros, los distintos aspectos de calidad de datos de interés para el escenario específico de cada organización" (Computación y Agrimensura, 2020). Esta tarea se organiza generalmente como un proceso iterativo que debe repetirse periódicamente. Apunta a monitorear los datos utilizados a lo largo del tiempo, ya que pueden haber cambios en los requerimientos o errores durante el relevamiento de los datos.

La calidad de datos se asocia generalmente con un conjunto de **dimensiones** que "agrupan propiedades o características de calidad" (Scannapieco y Catarci, 2002). Dentro de estas dimensiones, se pueden definir los **factores**, que representan "aspectos particulares de una dimensión de calidad (p. ej. correctitud sintáctica)" (Scannapieco y Catarci, 2002). Para cada factor se tiene luego una serie de **métricas**, que son "instrumentos que definen la forma de medir un factor de calidad (p. ej. porcentaje de datos sintácticamente correctos)"(Scannapieco y Catarci, 2002). Estas pueden ser genéricas o específicas, donde las métricas genéricas representan un conjunto de métricas con uso similar, que pueden ser refinadas a métricas más específicas. Para definir una métrica es necesario especificar su semántica (qué mide), granularidad

(sobre qué mide) y el tipo de resultado (p. ej. porcentaje o booleano). Una vez definida una métrica, es posible instanciarla especificando sobre qué elementos de la base de datos, como una columna o una tabla en particular, se aplicará. Las granularidades pueden variar, y una lista de las distintas granularidades (y su correspondencia con el modelo relacional) definidas en el marco de referencia se presenta en la Tabla 2.1. Para cada métrica se pueden definir, finalmente, distintos **métodos de medición**, que son "procesos que la implementan y se encargan de realizar medidas para la misma" (Scannapieco y Catarci, 2002). Estos modelos y sus relaciones se pueden observar en la Figura 2.1.

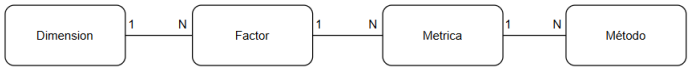

Figura 2.1: Modelo conceptual de los objetos del marco teórico

Algunas dimensiones frecuentemente consideradas para calidad de datos son (Christopher Fox y Redman, 1994):

- **Exactitud**: Mide qué tan cercanos están los datos actuales con respecto a sus valores esperados en el mundo real.
- **Consistencia**: Mide qué tan correcto es un dato en relación con las reglas semánticas que se esperan de él en el contexto del mundo real.
- **Completitud**: Mide el nivel de precisión con el que el dato ha sido modelado en comparación con su referencia. Evalúa si incluye todas las dimensiones necesarias para cumplir adecuadamente con las tareas previstas.
- **Unicidad**: Mide cuántas representaciones existen para un mismo dato del mundo real. Idealmente, dicho dato debería tener una única representación.
- **Frescura**: Mide la rapidez con la que los cambios en el mundo real se reflejan en los datos, es decir, con qué frecuencia estos son actualizados.

Granularidad	En Modelo Relacional
instanciaAtributo	celda
atributo	columna
conjuntoAtributos	conjunto de columnas
instanciaEntidad	tupla
entidad	tabla
conjuntoEntidades	conjunto de tablas
colección	base de datos
conjuntoColecciones	conjunto de bases de datos
organización	organización
conjuntoOrganizaciones	conjunto de organizaciones

Tabla 2.1: Tipos de Granularidad

En un proceso de gestión de calidad de datos, el modelo de calidad de datos abarca desde dimensiones hasta métodos de medición, e incluye a que datos se le aplican estos métodos. Funciona entonces como una guía para la gestión que se realiza sobre los datos del escenario al que se aplique.

2.2. Marco para la Gestión de Calidad de Datos

En esta sección se hace foco en dos componentes clave para la aplicación del marco de referencia publicado por AGESIC (Computación y Agrimensura, 2020). El primero es un Proceso de Gestión de Calidad de Datos que detalla distintas etapas a seguir y los distintos roles involucrados en llevarlas a cabo. El segundo es un Modelo de Calidad de Datos de Referencia, basado en los conceptos de modelos de calidad de datos descritos en la sección anterior, para usar como base en la definición de nuevos modelos para escenarios específicos.

2.2.1. Proceso de Gestión de Calidad de Datos

El marco de referencia sugiere la conformación de un Comité de Calidad de Datos, con un conjunto de roles que lo integran para llevar a cabo distintas tareas vinculadas a la gestión de la calidad de datos en una organización. Los roles destacados son los siguientes (Computación y Agrimensura, 2020):

1. **Responsable de Calidad de Datos (International, 2017)**: Este rol esta a cargo de dirigir el proceso, y de supervisar a los otros miembros del comité.

2. **Analista de Calidad de Datos (International, 2017)**: Es el encargado de relevar la información necesaria para crear el modelo de calidad de datos. Esto puede ser definir las dimensiones, factores y métricas, identificar potenciales problemas, o identificar las organizaciones y actores involucrados en el proceso.

3. **Técnico de Calidad de Datos**: Se ocupa de gestionar los recursos técnicos usados en el proceso de calidad. Se encarga, por ejemplo, de definir los métodos de medición.

4. **Experto de Negocio (International, 2017)(Wende, 2007)**: Este rol no está involucrado directamente en el manejo del modelo de calidad, pero trabaja en conjunto con los analistas, técnicos y el responsable de calidad de datos para brindar información sobre el negocio cuyos datos se desean medir. Su función es ayudar a definir los requerimientos de calidad de los datos y evaluar los resultados del proceso de calidad.

5. **Experto Técnico (Wende, 2007)**: Este rol se encarga del manejo de las distintas fuentes de datos y aplicaciones sobre los cuales se miden los datos. Es el referente técnico sobre el cual se apoya el resto del comité.

Para llevar a cabo el proceso de gestión, el marco de referencia establece siete etapas que deben ser ejecutadas. En cada una de ellas participan distintos equipos y roles, según la relevancia de su área de especialización respecto a las tareas correspondientes.

Como se puede observar en la Figura 2.2 el proceso comienza con la etapa de identificación de todos los elementos que componen el escenario de trabajo. A continuación, se analizan en profundidad dichos elementos para realizar una primera estimación de la calidad de los datos. Con base en esta evaluación inicial, se definen los requerimientos necesarios y se construye el modelo de calidad. Luego, se desarrolla la etapa central de este proyecto: medir y evaluar la calidad de los datos. En esta fase se aplican los métodos previamente definidos para generar información relevante y construir perfiles de evaluación. Una vez completada esta etapa, la información recolectada se utiliza para identificar las causas subyacentes de los problemas de calidad. Finalmente, la última etapa consiste en definir un plan de mejoras que permita abordar los problemas detectados en el escenario analizado.

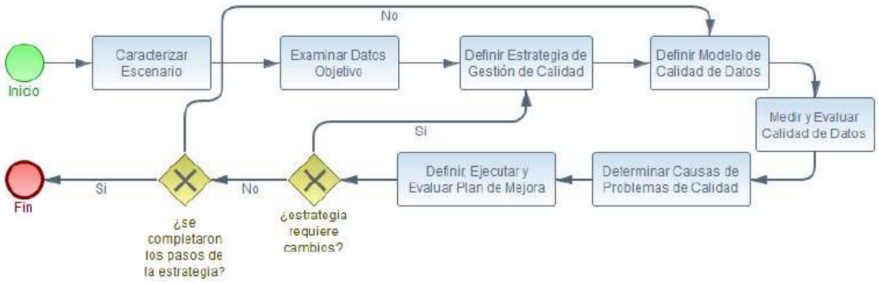

Figura 2.2: Proceso para la Gestión de la Calidad de Datos
(Computación y Agrimensura, 2020)

Para poder entender qué conlleva la quinta etapa del proceso, medir y evaluar calidad de datos, se definen dos conceptos adicionales: las reglas de evaluación y los perfiles de evaluación. Una **regla de evaluación** "especifica una condición que permite evaluar si el resultado de una medición para una métrica instanciada se encuentra (o no) dentro de los rangos de valores esperados" (Computación y Agrimensura, 2020). Los **perfiles de evaluación** son agrupaciones de estas reglas. Las reglas y perfiles de evaluación son usados una vez realizadas las mediciones, para determinar el estado de los datos según lo esperado.

Tal como se mencionó en los conceptos generales de calidad de datos, el proceso presentado debería ser cíclico. Como la estrategia puede cambiar regularmente, el proceso debería ejecutarse periódicamente para actuar en función a potenciales cambios en el escenario de trabajo.

2.2.2. Modelo de Calidad de Datos de Referencia

Si bien el enfoque de este trabajo es la quinta etapa del proceso, medir y evaluar la calidad de datos, no es posible realizar esta tarea sin resolver adicionalmente la etapa anterior de definición del modelo de calidad de datos. Para esto se toma como base el Modelo de Calidad de Datos de Referencia (Computación y Agrimensura, 2020) propuesto por el marco de referencia. El mismo está compuesto de una serie de dimensiones, factores y métricas generales, que pueden ser adaptados a escenarios variados.

Este modelo de referencia contiene las dimensiones previamente mencionadas, con métricas y factores para cada una. Por ejemplo, algunos de los factores que conforman la dimensión de Exactitud son la Correctitud Semántica y la Correctitud Sintáctica, las cuales representan la proximidad del valor de un dato con respecto a su valor real y su valor verdadero, respectivamente. Para Correctitud Sintáctica, un ejemplo de métrica es Formato que verifica que el valor de un atributo cumpla con el patrón o estructura esperada, como por ejemplo que una fecha siga el formato AAAA-MM-DD o que un código tenga una longitud y composición específica, ga-

rantizando que los datos respeten las reglas de formato definidas. También resulta ilustrativo el caso de la dimensión Consistencia, que aborda la coherencia interna y externa de los datos, e incluye factores como Integridad Inter-entidad, Integridad Intra-entidad, Integridad de Dominio y Consistencia Topológica. En particular, Integridad Intra-entidad captura la satisfacción de reglas entre atributos de una misma entidad. Una métrica asociada es ReglaIntegridadIntraEntidad, la cual permite identificar valores que, aunque válidos de forma individual, resultan incompatibles cuando se consideran en conjunto dentro del contexto de una misma instancia, como podría ser el caso de un paciente registrado como hombre que tiene asociada una patología que solo puede presentarse en mujeres.

Los resultados esperados de las métricas dependen de su nivel de granularidad, por lo que pueden presentarse como un resultado global —por ejemplo, un único porcentaje o un valor booleano— o como un resultado más detallado, que proporcione un valor para cada instancia de la entidad evaluada. En los casos donde el resultado de una métrica es booleano, también es posible definir métricas complementarias que midan el porcentaje de instancias que cumplen (o no) con la condición establecida. Estas métricas funcionan de manera complementaria: una ofrece una visión general rápida para facilitar una evaluación global, mientras que la otra proporciona un análisis detallado que permite identificar áreas específicas para mejorar y corregir.

2.3. Arquitecturas de Big Data

Las arquitecturas de big data (Tejada, 2021) son arquitecturas diseñadas para manejar la entrada, procesamiento, y análisis de datos que suelen ser demasiado grandes o complejos para sistemas de bases de datos tradicionales. Estas arquitecturas pueden variar en función a los recursos y tecnologías que tienen a disposición las distintas organizaciones y los usuarios que usan el sistema. Por ejemplo, mientras que se considera que estas arquitecturas suelen contar con grandes volúmenes de datos, el rango de lo que se considera mucho puede variar de organización a organización (siendo miles de gigabytes en algunos casos o miles de terabytes en otros).

Las arquitecturas de *big data* no se definen únicamente por tener muchos datos. Son también sistemas donde los datos pueden provenir de muchas fuentes distintas, donde son procesados con distintas operaciones para extraer valor para distintas áreas o son guardados en formatos o bases de datos heterogéneas (bases relacionales, no relacionales, archivos, etc.).

Algunas características comunes de estas arquitecturas son:

- **Fuentes de datos variadas**: los datos de las arquitecturas de big data suelen ser generados por fuentes de datos distintas, como bases de datos de aplicaciones, logs, archivos estáticos, APIs externas, o flujos de datos en tiempo real.
- **Almacenamiento de datos distribuidos y variados**: De la misma forma, los datos se guardan en distintos tipos de bases de datos, inclinándose en muchos casos al uso de bases no relacionales. Las fuentes usadas para procesa-

miento de datos se guardan muchas veces en archivos con formatos específicos que faciliten las consultas necesarias para el sistema. Este uso de distintos formatos hace que normalmente se tenga los datos distribuidos y en formatos diferentes.

- **Procesamiento por lotes**: Para manejar los grandes volúmenes de datos, las arquitecturas suelen soportar procesos que corren en segundo plano de la aplicación sobre largos períodos de tiempo.

- **Análisis e informes**: Las arquitecturas de big data suelen tener componentes orientados al análisis de los datos, y la generación de reportes de negocio que permitan sacar la información necesaria de los datos. Muchas veces son generados en procesos por lotes, como se mencionó en un punto anterior.

- **Procesamiento de Flujos de Datos**: En los casos donde se reciben datos de fuentes continuas, como por ejemplo controladores de herramientas meteorológicas, las arquitecturas deben contar con algún mecanismo que permita procesar un flujo continuo de datos. Muchas veces, estos módulos no solo reciben y guardan estos resultados, sino que además realizan operaciones adicionales sobre los mismos para procesar, filtrar y preparar los datos para el análisis.

- **Escalabilidad**: Estas arquitecturas suelen estar diseñadas para ser fácilmente escalables, ya sea para agregar nuevas fuentes de datos o módulos de procesamiento. Muchas veces, esto lleva a arquitecturas a base de eventos, que permiten que las actualizaciones se dispersen de forma asincrónica a través de los distintos componentes del sistema. Esto facilita la incorporación de nuevos componentes, ya que solo se los debe suscribir al canal de eventos.

- **Procesamiento en tiempo real**: Para operaciones más cortas, que requieren resultados más inmediatos, también se debería tener un módulo que permita computar datos en tiempo real.

En resumen, los componentes comunes de una arquitectura de Big Data se ilustran en la Figura 2.3. Si bien no todas las arquitecturas deben incluir necesariamente todos estos elementos, y algunas pueden incorporar componentes adicionales, el diagrama ofrece una visión general del flujo de información que gestionan este tipo de sistemas, así como de las formas habituales en que se distribuye la carga entre distintos módulos funcionales.

Existen herramientas y formatos que facilitan el manejo de datos en arquitecturas de big data. Por ejemplo, las siguientes librerías de Python orientadas al análisis de datos y al data profiling: Pyspark [1], Pandas [2] (con Geopandas para análisis de datos geográficos) y Great Expectations [3]. Por otro lado, los formatos columnares

[1] https://spark.apache.org/docs/latest/api/python/index.html
[2] https://pandas.pydata.org/docs/user_guide/index.html
[3] https://docs.greatexpectations.io/docs/

de Apache, Parquet y ORC [4], diseñados específicamente para grandes volúmenes de datos.

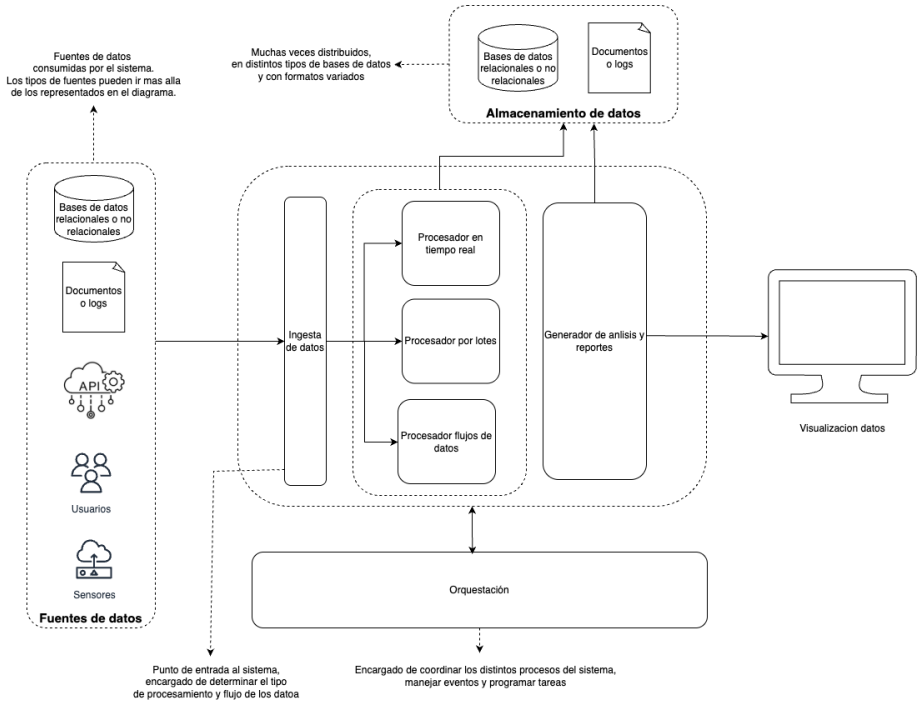

Figura 2.3: Diagrama de componentes comunes de las arquitecturas de Big Data (Tejada, 2021)

[4] https://onlinelibrary.wiley.com/doi/full/10.1002/cpe.5523

Capítulo 3

Revisión de Antecedentes

Con base en estas definiciones y conceptos básicos de calidad de datos, se investigaron las aplicaciones y herramientas existentes que pueden brindar apoyo durante el desarrollo del proyecto. En particular, se analizó el trabajo realizado en el proyecto de grado antecesor del presente proyecto, así como las aplicaciones disponibles en el mercado que sirvieron de inspiración para el diseño de la arquitectura.

3.1. Proyecto Fiumarelli y Fornio

La plataforma implementada en 2020 (Fiumarelli y Fornio, 2020) es una parte importante de este trabajo, ya que se busca extender el trabajo realizado en ese proyecto. Los componentes principales de esta aplicación son los siguientes:

- **Proceso de Gestión de Calidad**: da soporte a seis de las siete etapas del proceso de gestión de calidad, usando roles y modelos de calidad definidos a nivel global de la plataforma por un usuario Súper Administrador (con cierta flexibilidad). La plataforma **no da soporte** a Medir y Evaluar Calidad de Datos.
- **Escenarios de Trabajo en las Organizaciones**: la aplicación permite definir varias organizaciones, que trabajan de forma independiente en sus distintos procesos de calidad.
- **Comités de calidad**: Conjuntos de roles de calidad encargados del proceso de calidad. Estos roles son definidos dentro de la aplicación, salvo el Super Administrador (que es creado al inicializar la plataforma) y el Administrador de cada organización.
- **Gestión del Modelo de Calidad de Datos de Referencia**: Este modelo de referencia es definido por el Super Administrador, que maneja las distintas versiones del mismo.
- **Configuración de la Plataforma**: este elemento es usado por el Super Administrador para registrar organizaciones, configurar datos referenciales del framework y gestionar las diferentes versiones del modelo de calidad.

Estos componentes se agrupan en una arquitectura que cuenta con dos partes principales:

1. La entidad central es la que provee a las demás organizaciones de versiones del modelo de referencia, técnicas genéricas, tipos de colecciones y más datos referenciales.

2. Las organizaciones por su lado, "retroalimentan" estas "definiciones" que pueden ser de utilidad para otras organizaciones, por ejemplo sugiriendo extensiones al modelo de referencia.

Figura 3.1: Diagrama de arquitectura de la plataforma de Fiumarelli y Fornio (Fiumarelli y Fornio, 2020)

Como se ve en la Figura 3.1, la arquitectura cuenta con una entidad central encargada de gestionar un conjunto de organizaciones y de proporcionar recursos comunes a las mismas, como el modelo de calidad de datos de referencia. Cabe señalar, sin embargo, que los modelos de calidad de datos no incluyen métodos: la aplicación asume que dichos métodos ya se encuentran implementados por fuera de la plataforma. Esta arquitectura permite, entonces, que distintas organizaciones funcionen de forma independiente con sus propios escenarios de trabajo dentro de una misma aplicación, compartiendo solamente algunos recursos utiles para distintos escenarios.

La plataforma se implementó en Java, utilizando el entorno de desarrollo Eclipse Java EE IDE. Como sistema de gestión de bases de datos se utilizó PostgreSQL. Para la implementación de la interfaz web se utilizó HTML, CSS y JavaScript. También, librerías como Wicket, Bootstrap y JointJs.

Para entender el funcionamiento de la plataforma, se comparte a continuación un ejemplo de lo que sería un flujo básico de uso de la misma. El primer paso corresponde a la creación de las configuraciones previas al comienzo del uso. Estas son, asumiendo que el usuario Super Administrador ya fue creado:

1. Publicar versión inicial del modelo de referencia.

2. Configuración de los datos referenciales iniciales (tipos de colecciones, tipos de atributos, roles, etc.)

3. Alta de organización y Administrador, para que este pueda empezar a operar dentro de su organización.

El usuario administrador puede crear otros usuarios y asignarles roles dentro de su organización. Según los permisos otorgados, estos usuarios podrán acceder a las seis etapas del proceso de calidad que la aplicación acompaña. Además, la herramienta permite importar y exportar modelos de calidad, junto con sus elementos extendidos, en formato JSON. Estos archivos pueden ser reutilizados por otras organizaciones o integrados en otras aplicaciones.

3.2. Herramientas Existentes

Una vez analizada la aplicación anterior, a extender en este proyecto, se buscó investigar herramientas existentes que facilitaran el desarrollo de una nueva aplicación. Para esto, se investigan una serie de aplicaciones, librerías y formatos orientados al análisis y procesamiento de datos, y al manejo de grandes volúmenes de datos. Lo obtenido durante esta investigación fue usado como fuente de inspiración para el desarrollo de una nueva arquitectura, o directamente incorporado al proyecto.

3.2.1. Aplicaciones

Para facilitar el análisis y diseño de la aplicación, se buscaron herramientas existentes que tuvieran funciones similares a la aplicación a implementar. También se investigaron herramientas que pudieran ser útiles para la implementación de la misma. Para encontrar estas herramientas, se realizó una búsqueda en internet, utilizando los siguientes palabras claves

- Data quality tools
- Open source
- Data quality libraries
- Data cleaning tools
- Data cleaning scripts

Se tuvieron en cuenta tanto resultados web, como por ejemplo páginas de programas existentes, como papers (Val Pushkarev y Talburt, 2010) (Paulheim, 2016). El enfoque del análisis fue en las herramientas open source, ya que esto es un requerimiento no funcional importante, y se buscaba en principio poder aplicar algunos de los resultados encontrados. Sin embargo, también se tuvieron en cuenta las no open source, como fuente de inspiración para el análisis y diseño.

Se definieron los siguientes parámetros para evaluar la utilidad las herramientas encontradas:

- ¿Ofrece servicios que puedan ser consumidos por otra aplicación?

- ¿Permite consumir otros servicios, o agregar nuevos módulos?
- ¿Permite conectarse a fuentes heterogéneas? (bases relacionales, documentales, grafos, archivos de texto, etc.)
- ¿Maneja datos geográficos? Esta pregunta se incluye ya que en el modelo de calidad de datos de referencia, existen métricas para verificar por ejemplo si dos calles se cruzan, o si coordenadas geográficas se encuentran físicamente cerca.
- ¿Está bien documentada? ¿Tiene ejemplos de uso, y guías de instalación?

Estas preguntas fueron respondidas únicamente para las herramientas open source, ya que estas son las que posiblemente podrían ser reutilizadas. Estas herramientas se pueden definir en dos categorías: aplicaciones y librerías, las cuales son detalladas en las siguientes secciones. Se investigaron además formatos específicos para el manejo de grandes cantidades de datos, ya que estos podrían resultar útiles no solo como formatos de entrada de la aplicación, pero además como posibles herramientas para el manejo de la gran cantidad de datos que se podrían generar potencialmente con las mediciones de calidad.

Se presentan primero las aplicaciones open source, que incluyen a: DataCleaner (DataCleaner, 2019), Aggregate Profiler (Sourceforge, s.f.), Apache Griffin (Foundation, 2018), MobyDQ (Rolland, 2022), OpenRefine (OpenRefine, 2022), y Talend Open Studio (Inc., 2023). Se puede ver una recopilación de los resultados en la Tabla 3.1

Nombre	**Ofrece servicios que puede consumir otra aplicación**	**Permite agregar módulos**	**Permite conectarse a fuentes heterogéneas**	**Maneja datos geográficos**	**Bien documentada**
Data Cleaner	Si	Si	Si	No	Si
Aggregate Profiler	Si	No	Si	No	Si
Apache Griffin	Si	Si	No	No	No
MobyDQ	No	No	No	No	No
OpenRefine	Si	Si	No	No	Si
Talend Open Studio	Si	Si	Si	No	Si

Tabla 3.1: Comparación de aplicaciones existentes de calidad de datos

Se consideraron adicionalmente 4 aplicaciones no open source, Attacama ONE, Datamartist, Aperture Data Studio, e Informatica Data Quality. Estas aplicaciones fueron investigadas como fuente de inspiración para la arquitectura de este proyecto. Las ideas extraídas de estas aplicaciones fueron, por ejemplo, incluir un sistema de notificaciones y diseños para la interfaz de usuario. Se vieron además un conjunto de características interesantes relacionadas a la inteligencia artificial, como por ejemplo recomendación de métricas o detección de errores basado en fuentes externas. Estas características caen por fuera del alcance de este proyecto, pero serían un interesante trabajo a futuro y son discutidas más adelante.

Se puede ver, sin embargo, que ninguna de las aplicaciones existentes consideradas cumple todas las funcionalidades que se quieren abarcar en este proyecto, y además, ninguna cuenta con una API o forma sencilla de integración con otra aplicación. Ninguna permite manejar exactamente un modelo de calidad, en especial uno tan específico como el requerido por organizaciones de gobierno digital.

3.2.2. Librerías y Formatos

Como se mencionó previamente, existen librerías y formatos que facilitan el manejo de grandes volúmenes de datos. Para este proyecto se investigó en primer lugar Pandas (NumFOCUS, 2023), una librería open source de Python que permite analizar y manipular grandes cantidades de datos. Proporciona estructuras de datos y funciones para trabajar con datos estructurados, lo que facilita el manejo, la limpieza y el análisis de datos tabulares, como hojas de cálculo o tablas de bases de datos. Es particularmente interesante si se utiliza junto con GeoPandas (GeoPandas Developers, 2022), una extensión de la biblioteca que se utiliza para trabajar con datos geoespaciales. Permite la manipulación, análisis y visualización de datos que tienen una componente geoespacial, como mapas, formas de tierra, polígonos, puntos geográficos y más. El uso de Pandas y Geopandas tiene una ventaja importante. Pandas es una librería muy utilizada dentro de los ambientes de Python, y por lo tanto, tiene mucho soporte dentro de la comunidad y cuenta con integraciones con muchas librerías. Tanto Pyspark como Great Expectations cuentan con funciones que permiten usar funcionalidades de pandas, y Great Expectations hasta usa Pandas en su implementación. Se considera entonces una librería muy útil, que puede ser usada para implementar algunos de los métodos necesarios dentro de la aplicación.

Great Expectations (Expectations, 2023) es una librería, también de Python, que permite hacer data profiling para validar y documentar datos. Es muy sencilla de usar y permite validar datos mantenidos en bases tanto locales como remotas, y también data frames provenientes de la librería Pandas comentada anteriormente. Mientras que Pandas se utiliza principalmente para la manipulación y el análisis de datos, Great Expectations está diseñado para la validación de datos y el aseguramiento de su calidad. Es por lo tanto una librería más adecuada al problema presentado en este proyecto, aunque cabe aclarar que la misma puede ser usada junto con Pandas sin problema. Se considera entonces otra herramienta muy valiosa para el desarrollo de la aplicación. Se realizó una pequeña prueba técnica con esta librería para ver cómo sería implementar un método que valide que una columna tenga valores de tipo Integer, y además que estos valores se encuentren dentro de un rango.

Se notó que la implementación fue sencilla, y que los resultados estaban dentro de lo esperado.

Finalmente, se investigó la librería Pyspark (Spark, 2023), la API de Python para Apache Spark, un motor de análisis unificado open source para el procesamiento de datos a gran escala. Pyspark cuenta con una API de Pandas propia, y por lo tanto tiene funciones de Pandas adaptadas a los DataFrames Pyspark. Permite además pasar objetos de Pandas a Pyspark y viceversa, por lo que se pueden combinar las funcionalidades de las dos librerías fácilmente. Algunas de las funcionalidades ofrecidas por Pyspark relevantes al problema son:

- Crear conexiones a bases relacionales, y a bases MongoDB.
- Leer y escribir archivos Parquet, CSV y JSON, algunos de los formatos de interés para la aplicación
- Manejo de grandes volúmenes de datos. Aunque tanto Pandas como Pyspark son librerías populares para el procesamiento y análisis de datos en Python, Pyspark está mejor adaptada para el procesamiento de grandes volúmenes de datos ya que realiza el procesamiento de forma distribuida mediante la API de Apache Spark en vez de en memoria como lo hace Pandas.
- Pyspark cuenta con SQL Integrado, o sea, permite ejecutar consultas SQL directamente sobre los datos utilizando Spark SQL.

Luego, se investigaron en particular tres formatos ofrecidos por Apache, llamados Parquet y ORC (Vohra, 2016), y por otra parte un formato llamado RCFile. Estos formatos son populares, y en ciertos aspectos similares. Sin embargo, cada uno cuenta con particularidades que ofrecen ventajas para funcionalidades distintas.

Apache Parquet es un formato de archivo columnar, comprimido y en formato binario usado para el almacenamiento y la recuperación de datos, de forma eficiente. Sus algoritmos de compresión y codificación están pensados para manejar grandes cantidades de datos complejos. El formato columnar presenta varias ventajas, como compresión y descompresión más eficiente, y consultas más eficientes (las columnas se almacenan juntas y el motor de consultas puede omitir las columnas que no necesita para una consulta en particular). Tiene como principal desventaja, sin embargo, operaciones de escritura más costosas debido a la compresión y codificación. Este formato es muy popular, y cuenta con una gran comunidad. Está integrado con varias herramientas y librerías externas, incluyendo algunas de las mencionadas en la parte anterior. Pyspark, en particular, es capaz de leer archivos Parquet, y cuenta con funciones para llevar sus DataFrames a este formato.

Apache ORC es muy similar a Parquet. Pero, en este caso, mientras que los datos también son guardados en columnas, se cuentan con una serie de índices que permiten saltear porciones de datos innecesarias cuando se realizan consultas. Para implementar estos índices, se guarda la información en filas, llamadas stripes, que contienen la información de las columnas indexadas, como de qué posición a que posición van. Es por lo tanto un formato medianamente mixto entre columnar y en filas, aunque es mayoritariamente columnar. Utiliza algoritmos de compresión

como Zlib o Snappy, entre otros. Mientras que tiene entonces una excelente performance en operaciones de lectura, las de escritura sufren de las mismas limitaciones que ocurren en Parquet.

RCFile es un formato diseñado para procesamiento de datos en sistemas distribuidos. Utiliza un híbrido entre almacenamiento columnar y en filas, separando las tablas en grupos de filas, que a su vez se separan por columnas. Parquet y ORC tienen más funcionalidades, y se recomiendan en general sobre RCFile ya que los anteriores son útiles en más escenarios, especialmente cuando el rendimiento de la escritura es importante y la flexibilidad en el esquema es crucial. Una desventaja importante de este formato es que no cuenta con funcionalidades tan avanzadas en compresión y codificación. Por estas razones, este formato no fue analizado en profundidad.

Se puede ver un resumen comparativo de los formatos analizados en la Tabla 3.2. Comparando los formatos, se considera que Parquet es particularmente interesante en este caso debido al soporte de esquemas evolutivos, que puede ser muy útil en el proceso de análisis de calidad de datos. Además, el soporte de este formato en las librerías analizadas previamente facilitan su posible integración durante el desarrollo.

	Parquet	**ORC**
Formato de almacenamiento de datos	Columnar	Orientado a columnas, híbrido entre columnar y en filas.
Soporte comunitario	Extenso, con varias herramientas y librerías.	No tan utilizado en comparación
Caso de Uso Ideal	Mejor aplicado en casos con numerosas operaciones de lectura y escasas de escritura	Mejor aplicado en casos con numerosas operaciones de lectura y escasas de escritura. Orientado al uso con Hives.
Compresión	Incluye varios códigos, como Snappy, GZIP, deflate, y BZIP2. Snappy es el default.	Incluye varios códigos, como Snappy o Zlib
Soporte de Evolución del Esquema	Tiene soporte a cambios en el esquema sin tener que reescribir todo el dataset, y soporta tipos de datos complejos.	Limitada

Tabla 3.2: Comparación de formatos para grandes volúmenes de datos

Capítulo 4

Etapa de Análisis

Analizando las necesidades del proceso de calidad, y el funcionamiento de la plataforma ya disponible para su manejo, se definieron una serie de requerimientos para una nueva aplicación. El objetivo principal de esto es dar más soporte a la etapa cinco de la plataforma existente (Medir y evaluar calidad de datos). Se busca entonces que:

- La aplicación permita definir un modelo de calidad, llegando hasta sus métricas y la instanciación de las mismas.
- La aplicación soporte conexiones a distintas fuentes de datos, para poder ejecutar esas métricas.
- La aplicación automatice el proceso de medición, poniendo a disposición los resultados para ser usados a lo largo del proceso de calidad.

Para la definición de los requerimientos funcionales, es importante aclarar el concepto de *job*, entendido como una forma de programar la ejecución de un conjunto determinado de métricas junto con sus métodos asociados. Esta combinación conforma lo que se denomina un modelo de calidad instanciado. A partir de esta noción, se establecieron una serie de requerimientos funcionales y no funcionales, así como los distintos roles de usuario involucrados en su uso.

4.1. Roles dentro de la aplicación

En esta sección se presentan los distintos roles de usuario que intervienen en la aplicación.

- **Usuarios administradores**: Son los usuarios responsables del proceso de calidad, que tienen permisos administrativos sobre la plataforma. En particular, son los usuarios capaces de invitar otros usuarios y definir los roles que estos tienen asignados.

- **Usuarios expertos en calidad**: Corresponden a los usuarios capaces de definir modelos de calidad, y programar *jobs* para realizar las mediciones definidas en los modelos instanciados.
- **Usuarios técnicos**: Estos usuarios son una extensión sobre los usuarios expertos en calidad, pero que son además programadores. Son por lo tanto los usuarios capaces de definir métodos nuevos en el sistema.
- **Usuarios visualizadores**: Usuarios que solo pueden visualizar mediciones y resultados. Estos son usuarios no técnicos que solo necesitan hacer uso de los resultados.

Estos roles se derivan de los definidos en el marco de referencia. La Tabla 4.1 muestra un mapeo que refleja las asociaciones entre los roles de la aplicación, sus equivalentes en dicho marco, y un ejemplo representativo de cada uno dentro de una organización.

Rol dentro de la aplicación	**Rol dentro del marco de referencia**	**Ejemplo dentro de una organización**
Rol administrador	Responsable de Calidad de Datos	Gerente de datos y analítica
Rol experto en calidad	Analista de Calidad de Datos	Analista de datos Senior
Rol técnico	Técnico de Calidad de Datos	Administrador de base de datos
Rol visualizador	Experto de Negocio / Usuarios fuera del comité con interés en los resultados	Auditor interno Gerente de producto

Tabla 4.1: Correspondencia entre roles de la aplicación, marco de referencia y perfiles organizacionales

4.2. Requerimientos funcionales

En esta sección se presentan de forma resumida los requerimientos funcionales identificados y los roles que pueden acceder a cada uno de ellos. La versión completa, redactada como historias de usuario, se encuentra disponible en el Anexo C.

1. **Manejo de sesión**: Los usuarios de cualquier tipo deben poder iniciar y cerrar sesión en la aplicación. La autenticación se realiza mediante correo electrónico y contraseña.
2. **Manejo de usuarios**: Los usuarios administradores de la aplicación deben ser capaces de invitar nuevos usuarios a la aplicación, modificar usuarios de la aplicación (especialmente para cambiar los roles asignados a los mismos).
3. **Manejo de conexiones de datos**: Los usuarios administradores, técnicos y expertos en calidad deben poder crear, modificar y eliminar conexiones a fuentes de datos. Las fuentes de datos deben poder ser relacionales (PostgreSQL

o MySQL), no relacionales (MongoDB) o archivos (CSV o JSON). Al crear una nueva fuente de datos, debe quedar disponible su esquema. Cuando una conexión ya ha sido creada, debe ser posible refrescar su esquema para que cualquier cambio en la fuente de datos se vea reflejado en el sistema.

4. **Manejo de modelos generales**: Los usuarios administradores, técnicos y expertos en calidad deben poder crear, modificar y eliminar modelos generales (dimensiones, factores y métricas). La creación de modelos generales debe integrarse fácilmente con otros sistemas.

5. **Manejo de métodos**: Los usuarios administradores y técnicos deben ser capaces de crear, modificar y borrar métodos ejecutables en las conexiones de datos, que permitan medir las métricas definidas.

6. **Manejo de perfiles y reglas de evaluación**: Los usuarios administradores, técnicos y expertos en calidad deben poder crear, modificar y eliminar perfiles y reglas de evaluación, asociados a un modelo.

7. **Manejo de *jobs***: Los usuarios administradores, técnicos y expertos en calidad deben poder crear, modificar y eliminar *jobs* para poder programar las ejecuciones de un modelo instanciado. Los *jobs* deben poder ser ejecutados manualmente, o ser programados mediante un *cron* para ser ejecutados en diferido.

8. **Notificación sobre *jobs***: Los usuarios de cualquier tipo deben poder recibir notificaciones al finalizar la ejecución de un *job* para poder saber cuando están disponibles nuevos resultados.

9. **Manejo de resultados de *job***: Los usuarios de cualquier tipo deben poder acceder y eliminar los resultados de las distintas ejecuciones de los *jobs* para medir y comparar la calidad de los datos. Estos resultados deben poder verse sin procesar, en una tabla, o con algún perfil de evaluación del modelo aplicado.

En resumen, se pueden ver los requerimientos funcionales y qué roles tienen acceso a ellos de manera gráfica en la Figura 4.1:

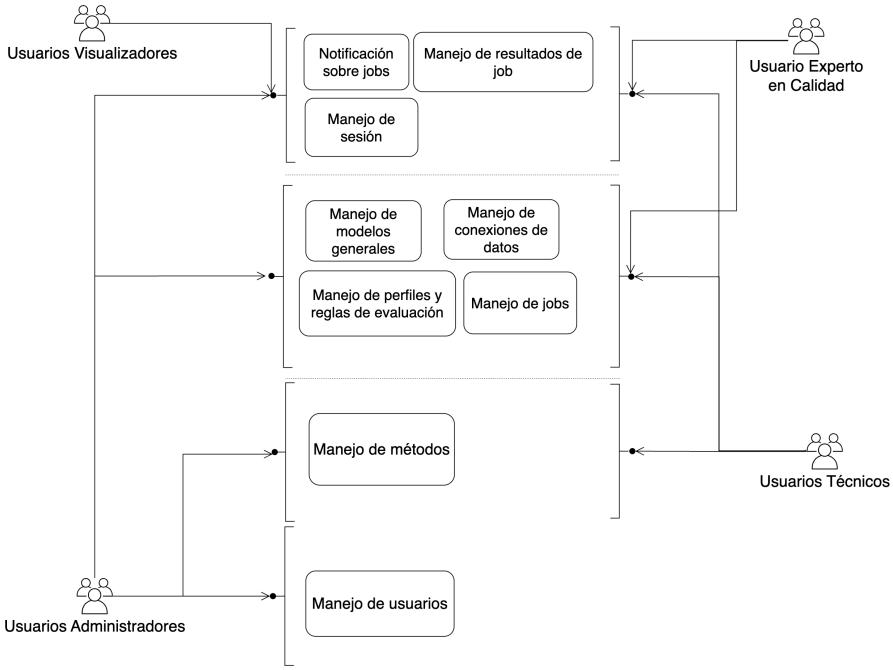

Figura 4.1: Diagrama de requerimientos funcionales

4.3. Requerimientos no funcionales

A continuación se presentan los requerimientos no funcionales identificados.

1. **Open source**: la aplicación implementada debe solo utilizar herramientas y librerías open source, debido a su contexto como aplicación para uso del gobierno digital.

2. **Heterogeneidad de las fuentes**: la aplicación deberá soportar conexiones a al menos una fuente de cada tipo: relacional, documental o basada en archivos.

3. **Extensibilidad**: la aplicación debe permitir agregar o reemplazar componentes sin modificar el código central de la solución, siguiendo una arquitectura modular o basada en plugins. Esto facilita, por ejemplo, el reemplazo del componente encargado de ejecutar métodos si se encuentra una tecnología más adecuada.

Capítulo 5

Solución Propuesta

A partir de los objetivos planteados y del análisis de antecedentes y requerimientos, se desarrolla una serie de métodos orientados a resolver los desafíos técnicos identificados, así como los principales componentes que conforman la arquitectura de la solución propuesta.

5.1. Descripción General

La aplicación fue diseñada siguiendo una arquitectura modular compuesta por tres grandes subsistemas: el portal web, el componente de lógica de negocio y el componente de procesamiento de datos. Cada uno de estos componentes cumple un rol específico en la solución, y se comunican entre sí mediante APIs REST, garantizando una interacción desacoplada y facilitando la evolución independiente de cada módulo. La Figura 5.1 muestra un diagrama general de la arquitectura, con sus distintos componentes y sus conexiones. La arquitectura contempla también la interacción de distintos tipos de usuarios, quienes acceden a funcionalidades específicas según su rol, a través del portal web.

Este diseño busca maximizar la reutilización de componentes, permitir su fácil reemplazo, y facilitar la integración con otras aplicaciones. La independencia entre módulos resulta especialmente útil en contextos donde es necesario adoptar nuevas tecnologías o adaptar el sistema a distintos tipos de fuentes de datos. Además, el sistema se apoya en dos bases de datos separadas: una para los objetos de negocio y otra para los resultados de calidad, lo que refuerza la separación de responsabilidades dentro de la aplicación.

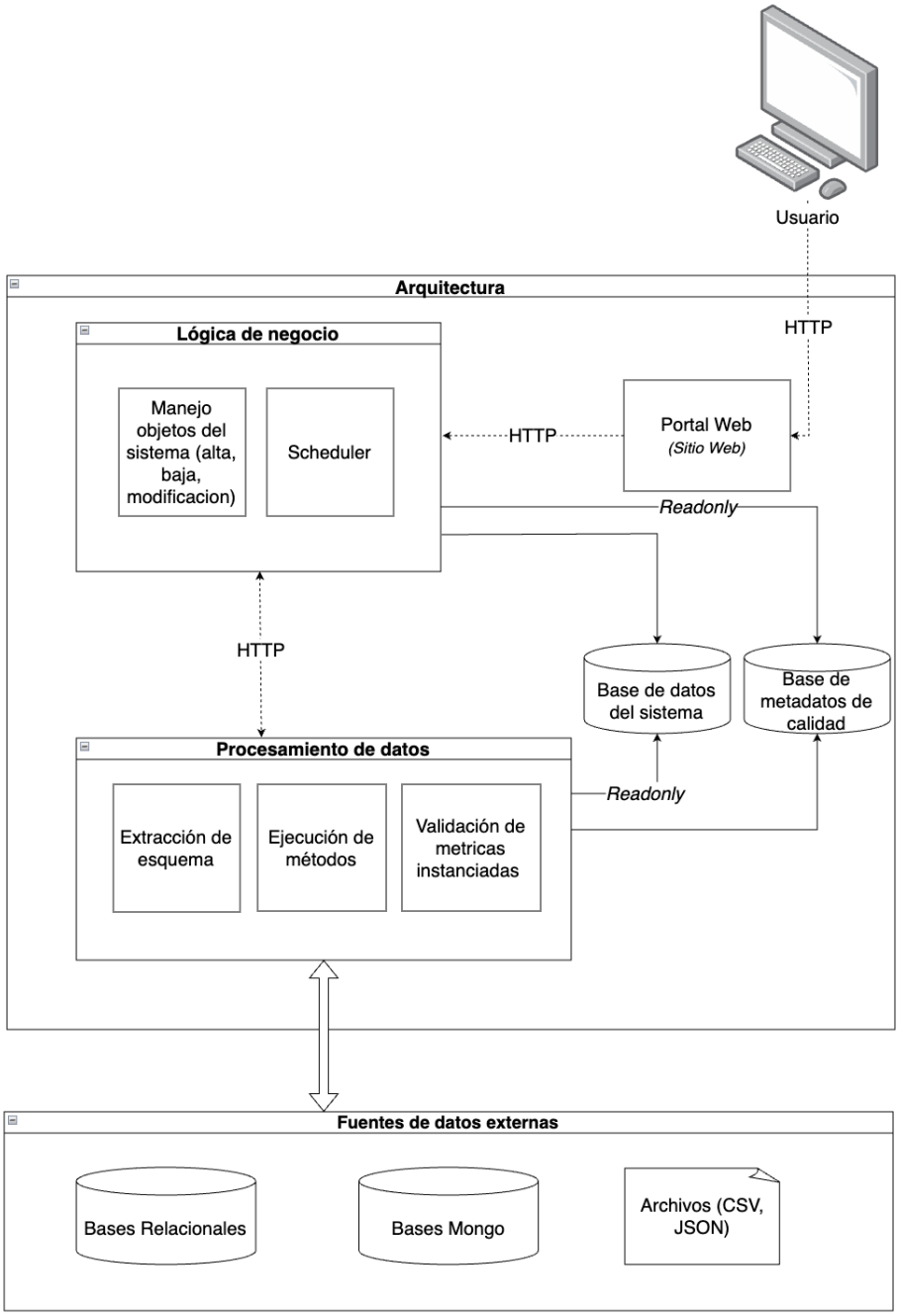

Figura 5.1: Diagrama de arquitectura del sistema

Con esta arquitectura, los componentes son fácilmente reemplazables. El único requisito es que implementen o consuman las APIs de los componentes con los que se relacionan. Esto es particularmente útil con el componente de procesamiento de datos, ya que si en algún momento este componente se quiere reemplazar por otro componente que use tecnologías más eficientes o tecnologías que se adapten mejor a otra fuente de datos, esto se puede hacer manteniendo el resto del sistema como estaba antes.

5.2. Principales Componentes

A continuación, se describen en mayor detalle los tres componentes principales que conforman la arquitectura de la aplicación, destacando el rol específico que cumple cada uno dentro del sistema y cómo se relacionan entre sí para dar soporte al proceso de evaluación de la calidad de datos.

5.2.1. Portal Web

El portal web constituye la interfaz principal a través de la cual los usuarios interactúan con la aplicación. A través de esta aplicación, accesible mediante un navegador, los distintos perfiles de usuario pueden gestionar modelos de calidad, configurar conexiones a fuentes de datos, visualizar resultados de evaluaciones y programar ejecuciones. El portal se comunica exclusivamente con el componente de lógica de negocio mediante una API REST, lo que garantiza un desacoplamiento que facilita tanto la evolución del frontend como la integración con otras interfaces en el futuro.

5.2.2. Lógica de Negocio

El componente lógica de negocio actúa como núcleo coordinador de la aplicación. Es responsable de orquestar las operaciones principales del sistema y gestionar los distintos objetos de negocio, tales como:

- **Modelos generales e instanciados**: definición, edición y consulta de dimensiones, factores, métricas y métodos.
- **Usuarios y permisos**: manejo de roles, autenticación y control de acceso.
- **Conexiones a fuentes de datos**: registro, actualización y eliminación de orígenes de datos heterogéneos.
- ***Jobs* y ejecución programada**: programación y ejecución manual o automática de modelos instanciados, a través de un *scheduler* interno.
- **Notificaciones y resultados**: coordinación del envío de alertas y acceso a los resultados de evaluaciones.

Este componente expone una API REST que sirve como punto de entrada tanto para el portal web como para el componente de procesamiento de datos. Su diseño modular permite desacoplar la lógica de negocio de los detalles de presentación y procesamiento, facilitando su mantenimiento y evolución.

5.2.3. Procesamiento de Datos

El componente de procesamiento de datos es el encargado de ejecutar los métodos definidos en los modelos de calidad sobre las distintas fuentes de datos configuradas en el sistema. Este componente cumple un rol clave en la etapa de evaluación, ya que se ocupa de:

- **Acceder a las fuentes de datos**: conecta con bases de datos relacionales (como PostgreSQL o MySQL), documentales (como MongoDB) y archivos planos (como CSV o JSON).
- **Extraer y actualizar esquemas**: permite consultar el esquema actual de una fuente y actualizarlo en el sistema cuando se detectan cambios.
- **Ejecutar métodos de evaluación**: aplica métodos sobre los datos para generar resultados, considerando la granularidad correspondiente.
- **Registrar resultados**: almacena los resultados en la base de metadatos de calidad, junto con la información necesaria para su trazabilidad.
- **Responder a solicitudes de ejecución**: recibe y atiende las solicitudes del componente de lógica de negocio mediante una API REST.

Gracias a su diseño desacoplado, este componente puede ser reemplazado o extendido fácilmente, por ejemplo, para incorporar nuevas tecnologías de procesamiento o integrar motores de ejecución especializados, sin afectar el resto de la aplicación.

5.2.4. Bases de Datos internas

Los componentes de lógica de negocio y procesamiento de datos operan utilizando dos bases de datos relacionales independientes entre sí.

La primera es la base de datos principal del sistema, que almacena los datos relacionados con los objetos de negocio, como dimensiones, factores, métricas, métodos, métricas instanciadas, *jobs*, reglas y perfiles de evaluación, usuarios, entre otros. El componente de lógica de negocio es responsable de la creación, modificación y eliminación de estos objetos. Por su parte, el componente de procesamiento accede a esta base únicamente en modo lectura, con el objetivo de obtener los datos necesarios para la ejecución y validación de los métodos.

La segunda base, llamada base de metadatos de calidad, contiene la información relacionada con las ejecuciones de las métricas instanciadas, sus resultados y los resultados de la aplicación de las reglas de evaluación. En este caso, el acceso es inverso: el componente de procesamiento es quien gestiona la escritura de los datos, mientras que el componente de lógica de negocio puede acceder a ellos en modo lectura para su visualización en el portal web.

Un aspecto particular de esta base de resultados es que, cuando se selecciona un método que devuelve resultados por fila, lo cual es especialmente útil en métricas con granularidad a nivel de “instancia de entidad” o de “instancia de atributo”,

se implementa un mecanismo para evitar la saturación de la base. En cada nueva ejecución de una métrica instanciada, los resultados generados previamente son eliminados, permitiendo así almacenar únicamente los datos más recientes.

5.3. Soporte a Fuentes Heterogéneas

Las conexiones a fuentes de datos fueron modeladas de forma independiente a su tipo de tecnología. El objetivo de este enfoque es permitir que los usuarios puedan ejecutar métodos definidos de manera global sobre cualquier tipo de fuente de datos, sin importar su naturaleza (relacional, documental, etc.). Para representar estas conexiones dentro del sistema, se adoptan los modelos definidos por el Marco de Trabajo de AGESIC (Computación y Agrimensura, 2020), que contemplan una estructura basada en Colecciones, Entidades y Atributos. En el caso de bases de datos documentales, donde pueden existir estructuras anidadas, las subentidades son representadas como entidades relacionadas con una entidad padre, bajo el concepto de SubEntidades. El sistema es capaz de aceptar, entonces, conexiones a los siguientes tipos de fuentes de datos:

- Bases relacionales MySQL
- Bases relacionales PostgreSQL
- Bases no relacionales de MongoDB
- Archivos CSV
- Archivos JSON

5.4. Métodos Personalizados

Los métodos son una parte central del proyecto, y son los responsables de calcular los valores de las métricas definidas en los distintos modelos. Son, sin embargo, independientes de las métricas y están diseñados para ser lo más generales posible, pudiendo correr en conexiones de PySpark independientemente a las fuentes de datos a la que se aplican. Para instanciar una métrica, entonces, se asocian:

- Una métrica
- Un método
- Una serie de parámetros de entrada (inputs)
- Una serie de sujetos definidos para el método.

Se introducen aquí dos conceptos fundamentales a los métodos, los **parámetros** y los **sujetos**.

Los parámetros se podrían considerar como constantes que se declaran dentro del método y pueden ser accedidos en el codigo del mismo. Pueden ser de tipo number, string, boolean o arreglo de alguno de los tipos ya mencionados. Para simplificar, el sistema no permite tipos más complejos como objetos, funciones, o arreglos de arreglos. Un ejemplo del posible uso de los parámetros es definir un array de valores posibles que podría tener un atributo, en el caso de un método que mide la correctitud semántica.

Los sujetos funcionan como variables, también accesibles dentro del código del método. Representan los distintos objetos accesibles dentro de una conexión, y pueden ser entonces colecciones, entidades o atributos.

Los sujetos y parámetros de un método se definen al crear el método, y son asignados para una métrica específica cuando esta es instanciada. Se accede a estos dentro del código usando el nombre de variable que se les asigna al crear el método. Los sujetos tienen acceso a una serie de funciones que permiten acceder a los datos de la conexión a los que se les asocian. Si se tiene entonces, por ejemplo, un sujeto de tipo atributo llamado 'att', se podría llamar a función del fragmento de Código 5.1 dentro del código del método para conseguir el nombre de la entidad a la que pertenece.

```
        tableName = att.getEntityName()

```

Código 5.1: Función para conseguir nombre de la entidad a la que pertenece un atributo

El código de un método puede estar implementado ya sea en SQL o en Python. En los casos donde se usa Python, los sujetos tienen funciones adicionales para acceder a la sesión de Pyspark y poder hacer uso de sus funcionalidades de análisis de datos.

El resultado de los métodos debe ser asignado, tanto en Python como en SQL, en una variable llamada 'result' que el sistema es capaz de detectar automáticamente.

Un ejemplo de una posible implementación para la métrica 'Ratio Correctitud Semántica' en Python se puede ver en el fragmento de Código 5.2

```
    tableName = col.getEntityName()
    createTempView(sparkSession, col.getConnection(), tableName,
    col.getEntitySchema())


    totalData = sparkSession.sql(f"SELECT count(*) as res FROM {
    tableName}")
    semanticallyCorrectData = sparkSession.sql(f"""
       SELECT count(*) as res
       FROM {tableName}
       WHERE {col.getAttributeName()} IN ({', '.join([f'"{value}"'
    for value in validValues])})
    """)


    result = (semanticallyCorrectData.first().res) / (totalData.
    first().res)
```

```
14
```

Código 5.2: Metrica implementada en Python

Los métodos le aportan mucha flexibilidad al sistema, pudiendo no solo definir código específico a cada proceso de calidad, pero también pudiendo acceder a APIs externas desde los métodos y a librerías de Python. Mientras requieren cierta comprensión sobre el funcionamiento de Pyspark, hacen que el sistema sea versátil. El Anexo A incluye el manual de usuario para la creación y el manejo de métodos. Contiene información detallada sobre todas las funciones ofrecidas por el sistema, y ejemplos de uso de los mismos.

5.5. Agenda de *Jobs* y notificaciones

La parte de la arquitectura responsable de correr los *jobs* programados por el usuario es el componente de procesamiento de datos. Para esto, la misma cuenta con una cola de *jobs*, que son procesados por lotes de forma a no afectar la responsividad del resto del sistema. El componente de lógica de negocio entonces envía *jobs* a la cola de ejecución ya sea cuando el usuario lo indica a través de la interfaz, o cuando el *cron* del "*Scheduler*" lo indica.

Los *jobs* tienen tres configuraciones adicionales además del *cron* para el *scheduler*: un límite de ejecuciones, configuración de las notificaciones y si el *job* está activado o desactivado.

El límite de ejecuciones del *job* cuenta la cantidad de veces que el *job* puede correr en total, contando las ejecuciones programadas y manuales. Si el *job* intenta correr de manera manual luego de haber llegado al límite de ejecuciones, la ejecución va a fallar. El límite de ejecuciones se puede dejar sin definir cuando se quiere que un *job* se corra indefinidamente.

Luego, con la configuración de las notificaciones, se puede determinar si se quieren recibir notificaciones dentro de la aplicación avisando cuando un *job* termina su ejecución. Las notificaciones contienen cuándo terminó la ejecución del *job*, su estado y detalles de los problemas que pueden haber ocurrido en casos de haber errores. Los posibles estados que puede tener la ejecución de un *job* son:

- **"Pending"**: Una nueva ejecución se encuentra en este estado cuando está esperando en la cola del componente de procesamiento de datos.
- **"Running"**: Una ejecución se encuentra en este estado cuando está corriendo y se están generando los resultados para cada métrica instanciada.
- **"Success"**: Una ejecución tiene este estado cuando terminó de correr, y todas sus métricas instanciadas se pudieron ejecutar correctamente.
- **"Failed"**: Una ejecución tiene este estado cuando terminó de correr, y todas sus métricas instanciadas fallaron al correr.
- **"Partially Failed"**: Una ejecución tiene este estado cuando termino de correr, y algunas de sus métricas instanciadas se pudieron ejecutar correctamente, pero otras fallaron.

- **"Failed To Run"**: Una ejecución tiene este estado cuando se intentó agregar a la cola de ejecución o se intentó correr, pero hubo algún error por el cual el *job* nunca se llegó a ejecutar.

Finalmente, cuando un *job* se desactiva, se suspenden temporalmente las corridas programadas por el "*Scheduler*". El *job* se puede activar o desactivar en cualquier momento. Si el *job* ya está corriendo, termina su ejecución actual, pero no permite enviar ninguna nueva.

5.6. Creación de Modelos

El modelo de calidad, tal como está definido en el marco de referencia, se representa en el sistema mediante la combinación de un modelo general y un modelo instanciado. El modelo general incluye exclusivamente las dimensiones, factores y métricas, funcionando como una abstracción reutilizable e independiente de cualquier fuente de datos específica. Por su parte, el modelo instanciado incorpora los métodos, los cuales se asocian a las métricas y se vinculan con distintas fuentes de datos. Esta separación permite reutilizar un mismo modelo general en múltiples contextos, aplicando distintos métodos y conectándolo a diferentes fuentes de datos. Gracias a esta flexibilidad, es posible definir un método una sola vez y luego reutilizarlo con métricas pertenecientes a distintos modelos generales, sin importar la fuente de datos sobre la que se aplique.

El sistema le ofrece tres formas al usuario de crear modelos generales:

- Usando el modelo de referencia
- Subiendo un archivo JSON con cualquiera de los dos formatos descritos en el Anexo B. Uno de los formatos es el definido por nuestro sistema, mientras que el otro es una versión adaptada del formato que exporta el proyecto de Fiumarelli y Fornio de 2020 (Fiumarelli y Fornio, 2020).
- Usando la interfaz web

Cabe destacar que si se usa una de las primeras dos opciones, de todas formas es posible acceder a la interfaz web una vez creado el modelo para poder realizar cambios sobre el mismo. La idea detrás de esto es que se pueda tomar el modelo de referencia o el modelo importado de otra aplicación y que el mismo pueda ser ajustado de forma manual a las necesidades del usuario.

Se decidió ofrecer esta vía de integración con otros proyectos para que la compatibilidad quede abierta a otras aplicaciones, no solo la de 2020. Cualquier proyecto capaz de generar un JSON con uno de los formatos esperados podría subir sus modelos a la aplicación.

5.7. Roles y usuarios

Para asegurar que los únicos usuarios que pueden acceder a la aplicación son aquellos de la organización, la aplicación no cuenta con una página de registro de

usuarios. En cambio, los usuarios deben ser invitados a la aplicación por un usuario administrador, que se encarga de manejar los usuarios con acceso a la aplicación y de asignarles roles a los mismos.

El primer usuario administrador debe ser creado directamente en la base de datos, y la aplicación debe contar siempre con al menos un usuario administrador. Por esta razón, un usuario administrador no puede borrarse a sí mismo del sistema, debe antes asignar otro administrador y este es el que podrá borrarlo.

Como se detalló previamente, los roles de los usuarios restringen las partes de la aplicación a las que tienen acceso los distintos usuarios. Fuera de los usuario administradores, que tienen acceso a toda la aplicación, los usuarios técnicos son los únicos que tienen acceso a crear métodos, los usuarios expertos en calidad son los responsables de definir los modelos de calidad y los *jobs*, y finalmente los usuarios visualizadores son solo capaces de ver los resultados de las ejecuciones de los *jobs*.

Los roles definidos dentro del sistema son menos de los roles de un posible comité de calidad, responsable de un proceso completo. Esto se debe a que esta aplicación se enfoca únicamente en una parte del proceso de calidad y no requiere tanta variedad de roles. Los roles definidos agrupan entonces los posibles roles del proceso de calidad.

Capítulo 6

Implementación y Verificación

Teniendo una imágen general de la solución propuesta para la aplicación, se continuó con el desarrollo de la misma. En esta etapa se definieron las tecnologías usadas para implementarla y los distintos flujos y sistemas mediante los cuales se interactúa con la misma.

6.1. Tecnologías utilizadas

El desarrollo de la aplicación se realizo en paralelo para el portal web, el componente de lógica de negocio y el componente de procesamiento de datos. Se utilizaron las siguientes tecnologías:

- **NextJS**: Marco web de desarrollo basado en React que facilita la creación de aplicaciones en javascript. Fue utilizado para el desarrollo del portal web de la aplicación.
- **NestJS**: Marco web de desarrollo progresivo para Node.js, construido con TypeScript. Fue utilizado para el desarrollo del componente de lógica de negocio.
- **Django**: Marco web de desarrollo para Python. Fue utilizado para el desarrollo del componente de procesamiento de datos, ya que es compatible con las herramientas de procesamiento de datos elegidas.
- **PySpark**: Utilizado en el componente de procesamiento de datos para manejar las conexiones a las fuentes de datos y operaciones sobre los mismos.
- **Docker**: Plataforma que facilita la creación, despliegue y ejecución de aplicaciones mediante el uso de contenedores virtualizados. Se tiene una configuración de Docker a nivel de cada componente de la arquitectura, y luego una configuración global capaz de levantar la aplicación en su totalidad.

- **MySQL**: Sistema de gestión de bases de datos relacional (RDBMS), utilizada por las bases de datos del sistema (base con la lógica de negocio y base de metadatos de calidad).
- **Elastic Cloud**: Plataforma de gestión de servidores utilizada para desplegar y administrar la aplicación creada.

Se utilizaron además las siguientes tecnologías para asistir el desarrollo:

- **Git, Github**: Sistema de control de versiones distribuido, que permite a varios desarrolladores trabajar en un proyecto de software de manera simultánea y coordinada, manteniendo un historial completo de todos los cambios realizados en el código fuente. Se utilizó GitFlow como modelo de ramificación para coordinar el trabajo entre integrantes y mantener un historial de *commits* claro, correspondientes a cada nueva funcionalidad.
- **Trello**: Herramienta de gestión de proyectos basada en la web. Se utilizó para ordenar el desarrollo y estar al tanto del trabajo del resto de los miembros del equipo.
- **Discord**: Plataforma de comunicación en línea que combina funciones de chat de texto, voz y video. Se utilizó como método principal de comunicación interna del equipo.
- **Google Drive**: Se utilizó una carpeta en la nube en la cuál se fue guardando la documentación generada a lo largo del desarrollo del proyecto, como resumen de material leído, diagramas, requerimientos, apuntes de reuniones, presentaciones, manuales de usuario, etc.
- **Figma**: Aplicación web que facilita el diseño de interfaces, ya sean para el navegador o móviles. Se utilizó para realizar el diseño inicial de la aplicación.
- **Postman y Swagger**: herramientas utilizadas para documentar las APIs desarrolladas por la aplicación. Ambas incluyen ejemplos de uso de dichas APIs.

6.2. Estructura de las bases de datos del sistema

Para comprender mejor cómo interactúan las dos bases de datos del sistema, es útil analizar en detalle los modelos que existen en cada una y las relaciones que se establecen entre ellos. Como se mencionó en el punto 5.3, las fuentes de datos se representan mediante un modelo basado en Colección, Entidad y Atributo. Los valores de estas tablas se generan a partir de la extracción del esquema correspondiente a cada Conexión con una fuente de datos. Esta estructura se encuentra modelada en la base de datos, tal como se ilustra en la Figura 6.1.

La extracción del esquema representa un desafío particular en el caso de fuentes no relacionales, debido a la heterogeneidad en sus estructuras. Más adelante se detalla cómo este problema fue abordado en el sistema y qué simplificaciones se implementaron para facilitar el proceso de extracción.

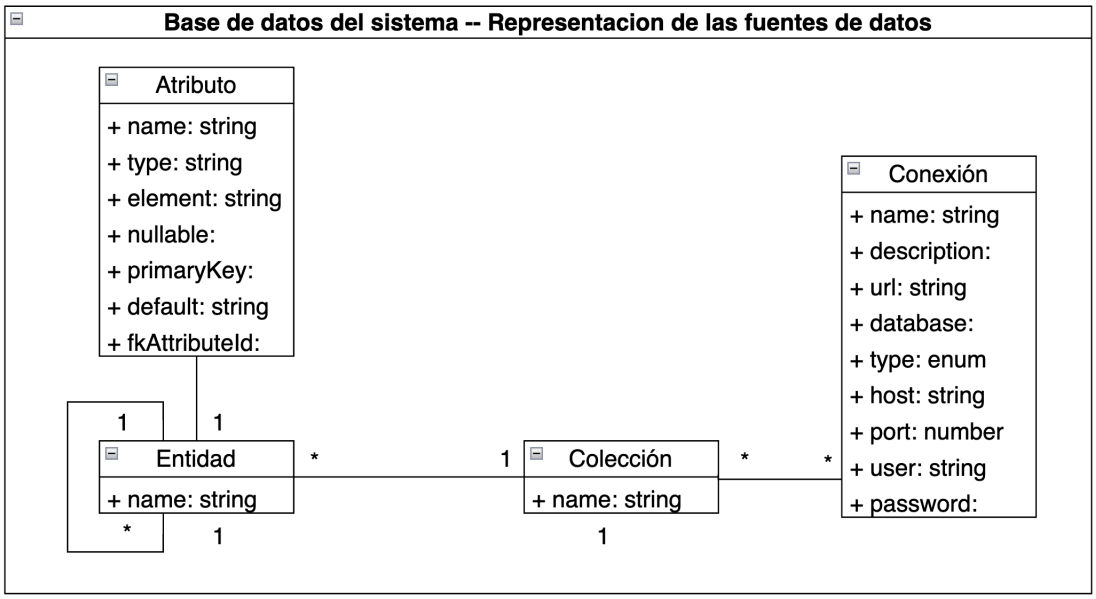

Figura 6.1: Representación de las fuentes de datos en la base de datos del sistema

Posteriormente, como se mencionó en el punto 5.6, el modelo de calidad de datos se divide en dos componentes: el modelo general y el modelo instanciado. Esta distinción también se refleja en la base de datos del sistema, cuya estructura se muestra en la Figura 6.2.

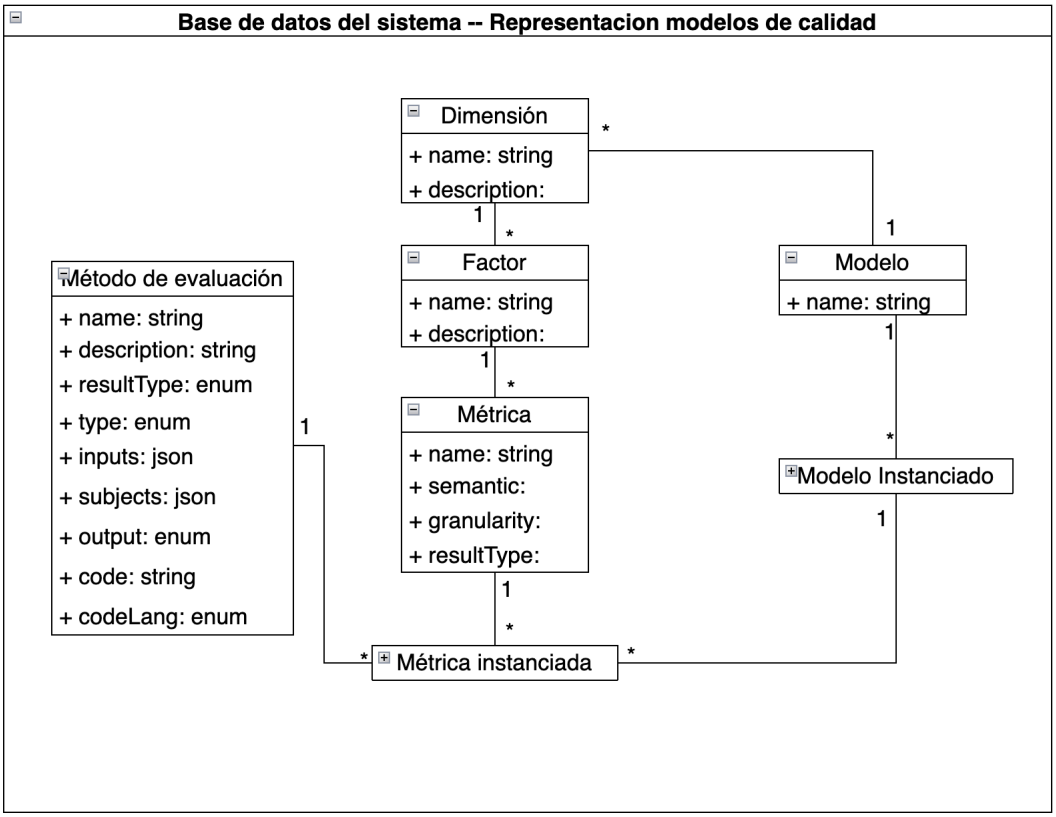

Figura 6.2: Representación de los modelos de calidad de datos general e instanciados

En el diagrama puede observarse cómo cada Modelo General puede tener asociadas múltiples instanciaciones. Estas instanciaciones representan la relación entre una Métrica Instanciada, un Método, y un elemento dentro de una conexión —ya sea una Colección, un Atributo o una Entidad—, a través de la noción de Sujeto del modelo. Esta estructura se ilustra en la Figura 6.3.

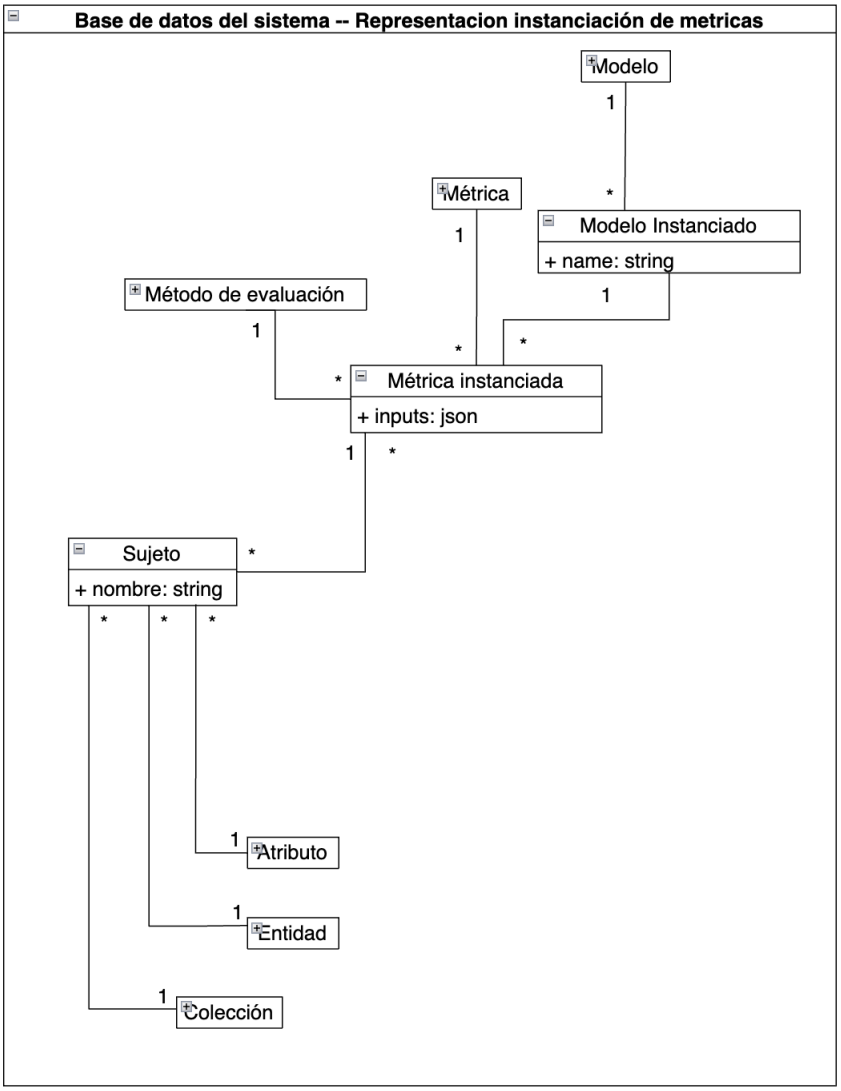

Figura 6.3: Representación de la instanciacion de un modelo general

Una vez instanciado el modelo, es posible ejecutar los métodos y aplicar los perfiles de evaluación mediante el uso de *Jobs*. Estos se encuentran vinculados al Modelo Instanciado y, por lo tanto, ejecutan todas las Métricas Instanciadas definidas en él. Esta dinámica se representa en la Figura 6.4. Es en este punto donde entra en juego la base de metadatos de calidad, ya que los resultados de las ejecuciones son almacenados en ella. En la Figura 6.5 se presentan las tablas correspondientes a dicha base, las cuales reflejan cómo se registran los resultados generados por el

sistema.

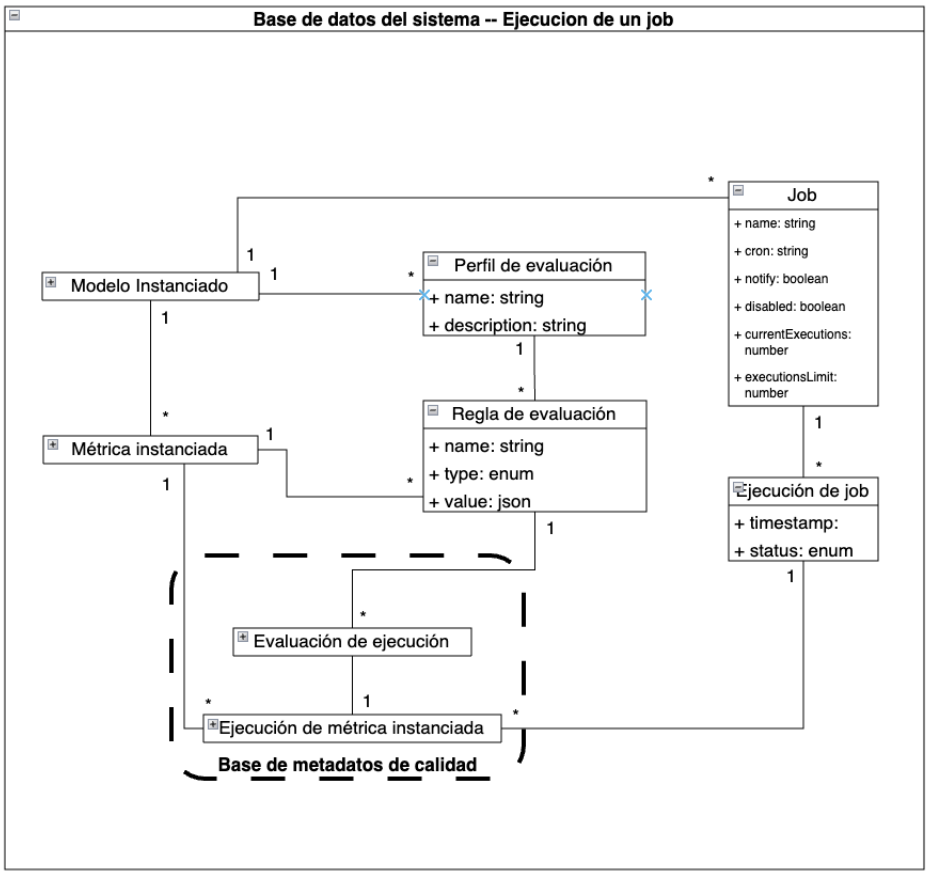

Figura 6.4: Representación los perfiles de evaluación y *jobs*

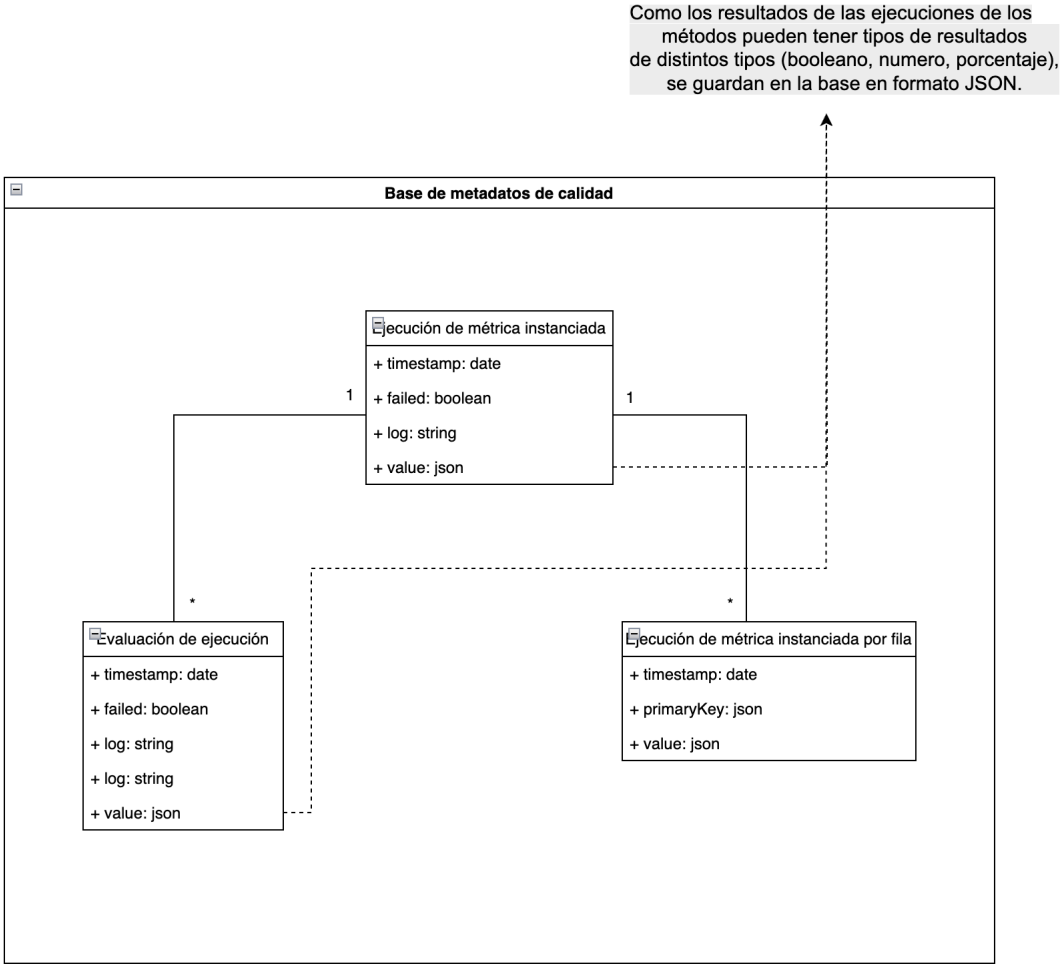

Figura 6.5: Diagrama de la base de metadatos de calidad

A nivel de cada *Job*, es posible definir si se desea notificar a los usuarios del sistema cada vez que se generen nuevos resultados. Las notificaciones pueden ser de distintos tipos, y tanto su contenido como su naturaleza dependen del resultado de la ejecución del *Job*. De este modo, los usuarios pueden conocer fácilmente si una ejecución fue exitosa o si se produjo algún error. La estructura del sistema de usuarios y del mecanismo de notificaciones puede observarse en la Figura 6.6. Además de las notificaciones dentro del sistema, los usuarios también reciben alertas por correo electrónico.

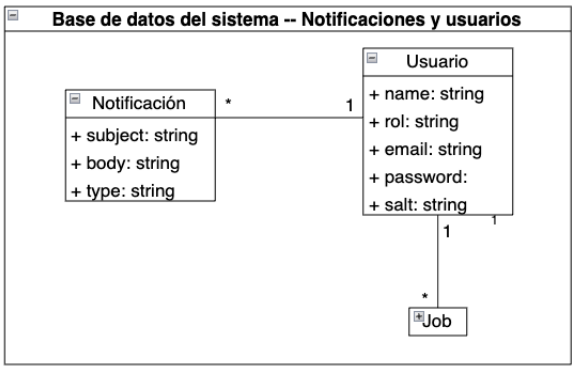

Figura 6.6: Representación de los usuarios y notificaciones

6.3. Extracción de esquema y acceso a fuentes heterogéneas

La extracción del esquema de una conexión depende del tipo de la fuente de datos. En el caso de las bases de datos relacionales, se utilizan las tablas de metadatos del sistema de gestión de bases de datos correspondiente, mientras que en el caso de MongoDB y las fuentes de datos basadas en archivos se utilizan funciones nativas de la librería PySpark. En este último caso, se planteó una problemática ya que ese tipo de fuente de datos, por definición, no tiene un esquema, tratándose de un problema abierto sin una solución única.

Se tomaron entonces las siguientes decisiones para simplificar el problema:

1. **Detección incremental de campos**: Si algún campo aparece en alguno de los documentos de las bases documentales, el mismo es agregado al esquema con el tipo con el que aparece la primera vez que es encontrado. Este comportamiento es “umbrella“, lo que significa que el primer tipo de dato observado es el que se usa en el esquema, sin importar si el campo puede tener otro tipo de dato en otros documentos. No se revisan las posibles variaciones del tipo de dato para ese campo en otros documentos.

2. **Información adicional de los datos**: Como no se cuenta con metadatos en estos tipos de fuentes, no se guarda información sobre las entidades y atributos como claves primarias, valores por defecto, posibilidad de valor nulo, etc. Cabe aclarar que esto también sucede en las conexiones de tipo CSV. En los casos de MongoDB, se le auto asigna el valor de clave primaria a la columna “_id“ presente en todos los documentos. Puede suceder que ciertos métodos requieran que las fuentes de datos cuenten con claves primarias. Para estos casos, el sistema permite que el usuario seleccione un atributo de forma manual para ser usado como clave primaria.

3. **Limitación de la profundidad**: Se limita la profundidad a la que llegan las *subentidades* del sistema en función a una variable de entorno para facilitar el desarrollo y la implementación de métodos del lado del usuario. La variable de entorno, sin embargo, ofrece flexibilidad al usuario en caso de que se necesite más profundidad que el valor por defecto (3 niveles).

Una vez extraído un esquema, el mismo es traducido al modelo de colecciones, entidades, subentidades y atributos, y esto es guardado en la base de datos del sistema. Para las bases de datos que no son archivos, existe la posibilidad de refrescar el esquema de la base para poder reflejar cualquier cambio que ocurra en la aplicación. Un detalle importante en estos casos es que si se tienen *jobs* y métricas instanciadas existentes al momento de actualizar las colecciones, las ejecuciones de los mismos pueden llegar a fallar parcial o completamente hasta que las métricas instanciadas afectadas sean actualizadas.

Para las bases CSV o JSON es posible subir archivos para refrescar los datos, pero no se puede actualizar el esquema. Esto requiere una conexión nueva.

Tanto para la extracción del esquema como para el uso de los datos por parte de los métodos, el acceso a las fuentes de datos se realiza mediante PySpark, cargando los datos en objetos del tipo DataFrame. Esta estrategia ofrece a los usuarios la flexibilidad de ejecutar métodos utilizando SQL, Python o las funciones nativas que proporciona PySpark. La abstracción de las fuentes de datos a través de este enfoque se representa en la Figura 6.7.

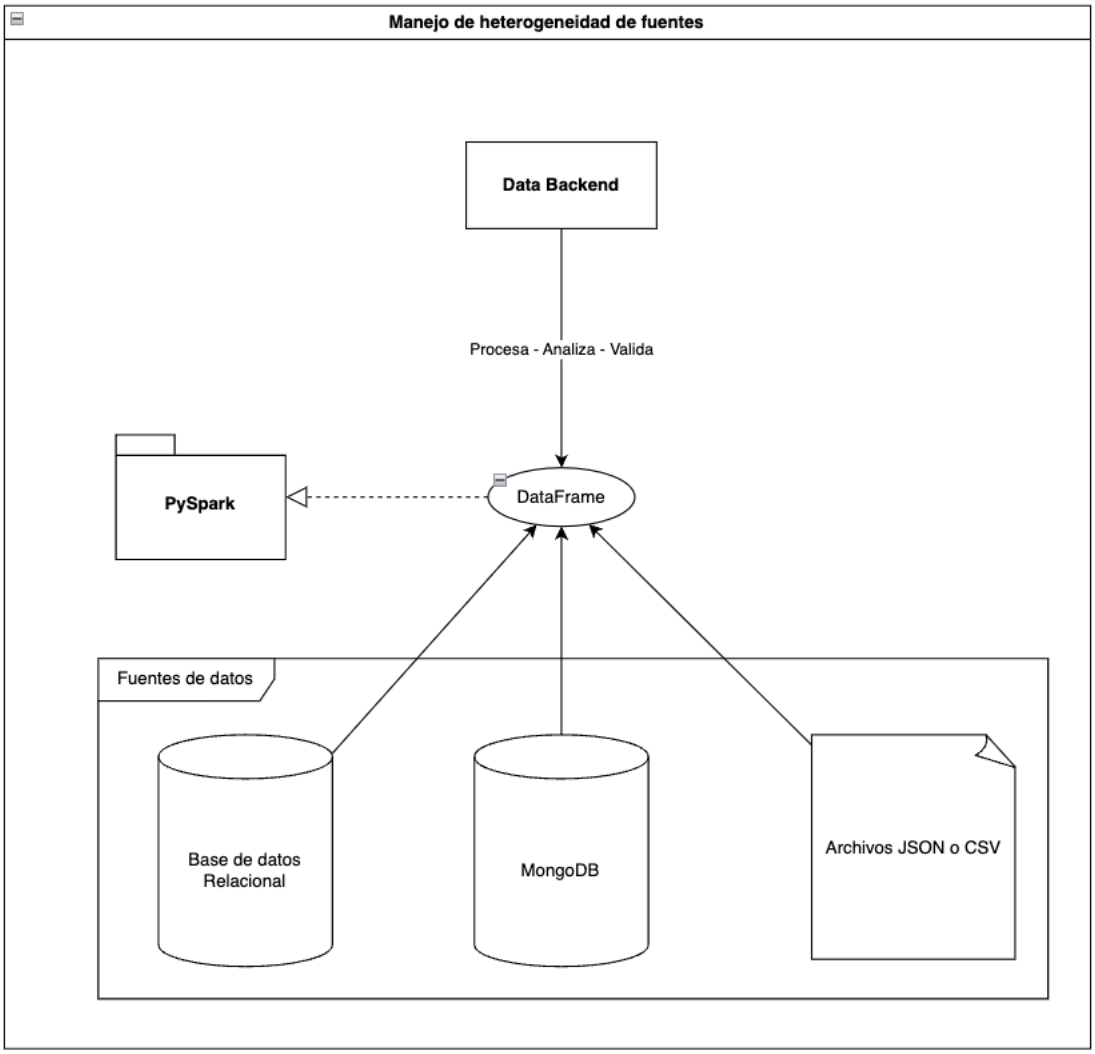

Figura 6.7: Diagrama de manejo de conexiones del sistema

6.4. Verificación

Una vez finalizada la implementación del sistema, se realizaron una serie de pruebas para verificar el correcto funcionamiento de los distintos componentes del sistema. Se dividieron en dos categorías:

1. **Pruebas manuales**: se realizó en dos instancias, a medida que se implementó el sistema, de forma local, y una vez que se levantó el sistema en un servidor poblando con los datos para el caso de uso. Estas pruebas se realizaron ya sea en el navegador para el portal web, o usando Postman para simular las llamadas HTTP a los componentes de lógica de negocio y a los componentes de procesamiento de datos.

2. **Pruebas de extremo a extremo (E2E), automatizadas**: se generó un plan de pruebas para verificar todos los endpoints del componente lógica de negocio, haciendo llamadas al componente de procesamiento de datos. Para cada prueba de la tabla generada se anotó para qué *endpoint* es, una descripción del caso y el resultado esperado o error que debería ocurrir. Se crearon en total 171 pruebas, que alcanzaron un *coverage* del 80 % de las líneas del código del componente lógica de negocio.

Capítulo 7

Caso de Estudio y Validación

A lo largo del desarrollo del proyecto se llevaron a cabo diversas etapas de prueba del sistema, así como reuniones con distintos equipos para validarlo y recibir retroalimentación sobre posibles mejoras. Algunas de estas instancias ocurrieron al inicio del proyecto y durante el diseño de la arquitectura. Otras, a lo largo de la implementación, o durante la realización del caso de uso. La ultima instancia, fue una demo realizada al finalizar este caso de uso. Todas estas instancias resultaron sumamente enriquecedoras y aportaron valiosas ideas para mejorar el sistema. Si bien no todas pudieron ser incorporadas, todas fueron registradas para que puedan ser consideradas en futuras extensiones del sistema.

7.1. Presentación del diseño inicial de la arquitectura

Una vez elaborado un primer bosquejo del diseño de la arquitectura, se realizó una presentación en el Instituto de Computación (INCO) ante un equipo de docentes con conocimientos en calidad de datos y una estudiante de doctorado especializada en el área. En dicha instancia, se brindó un panorama general sobre el proceso de calidad de datos y se presentó la información recopilada durante la etapa de análisis de antecedentes. Posteriormente, se expusieron los requerimientos funcionales y no funcionales definidos hasta ese momento, así como el diseño preliminar de la arquitectura. Finalmente, se realizaron dos demostraciones utilizando las bibliotecas seleccionadas para la implementación del componente de procesamiento de datos, con el objetivo de validar la viabilidad del sistema.

De esta reunión surgieron valiosos aportes para mejorar el trabajo desarrollado hasta el momento, recibiendo especialmente apoyo en la parte del sistema encargada de extraer los esquemas de las fuentes de datos, con el fin de definirlas dentro del modelo del sistema. Este aspecto resulta particularmente complejo en el caso de bases de datos no relacionales, dado que no existe una solución única que se adapte a todos los escenarios. Por ello, la instancia resultó sumamente útil para delimitar el alcance del sistema y para identificar aquellas áreas donde era conveniente acotar el alcance para reducir la complejidad del desarrollo.

Además, la reunión fue una oportunidad para definir posibles formas de integración con otras aplicaciones, a partir de la identificación de componentes útiles para la propia aplicación y de propuestas de mejoras en la arquitectura, con el objetivo de facilitar su uso en conjunto con otros sistemas.

7.2. Caso de estudio

Para validar funcionalmente la aplicación, se llevó a cabo un caso de uso con datos simulados usando un modelo de calidad de datos desarrollado en el marco de una maestría en curso para un contexto real (Pidre, 2025). Este fue creado para la Plataforma Salud.uy, una plataforma tecnológica desarrollada para dar soporte al sistema de salud nacional, que se encarga de la gestión de la Historia Clínica Electrónica Nacional (HCEN). La plataforma tiene una arquitectura con varios componentes, y este caso de uso se enfoca en uno en particular: el Índice Nacional de Usuarios de Salud (INUS), que gestiona los datos patronímicos de los pacientes y tiene como finalidad identificarlos unívocamente dentro de la plataforma. El INUS utiliza un identificador global de pacientes y gestiona además su correspondencia con los identificadores locales que maneja cada Prestador de Salud. Por lo tanto, cuenta con datos sensibles, e información privada de la población, y por esta razón se decidió simular los datos en base a un mensaje de ejemplo que muestra qué campos contiene la base de datos del mismo. Los datos además fueron modificados para mejorar su legibilidad (originalmente, estaban en formato PIX, y por lo tanto los nombres de las columnas no eran descriptivos), para limitar la profundidad del esquema del JSON generado, y para incluir un ID en los distintos objetos que permitiera definir una clave primaria (necesario para los métodos de tipo instanciaEntidad e instanciaAtributo).

El modelo diseñado en este caso se basa en el modelo de datos general creado para la aplicación, pero cuenta con algunos cambios. Se extrajeron entonces en primer lugar las dimensiones, factores y métricas definidas por el modelo, y las mismas se crearon en la aplicación como un modelo general. El modelo general llega hasta las métricas, pero no cuenta aún con métodos que las implementen. Se creó un modelo con la estructura que se presenta en la Figura 7.1.

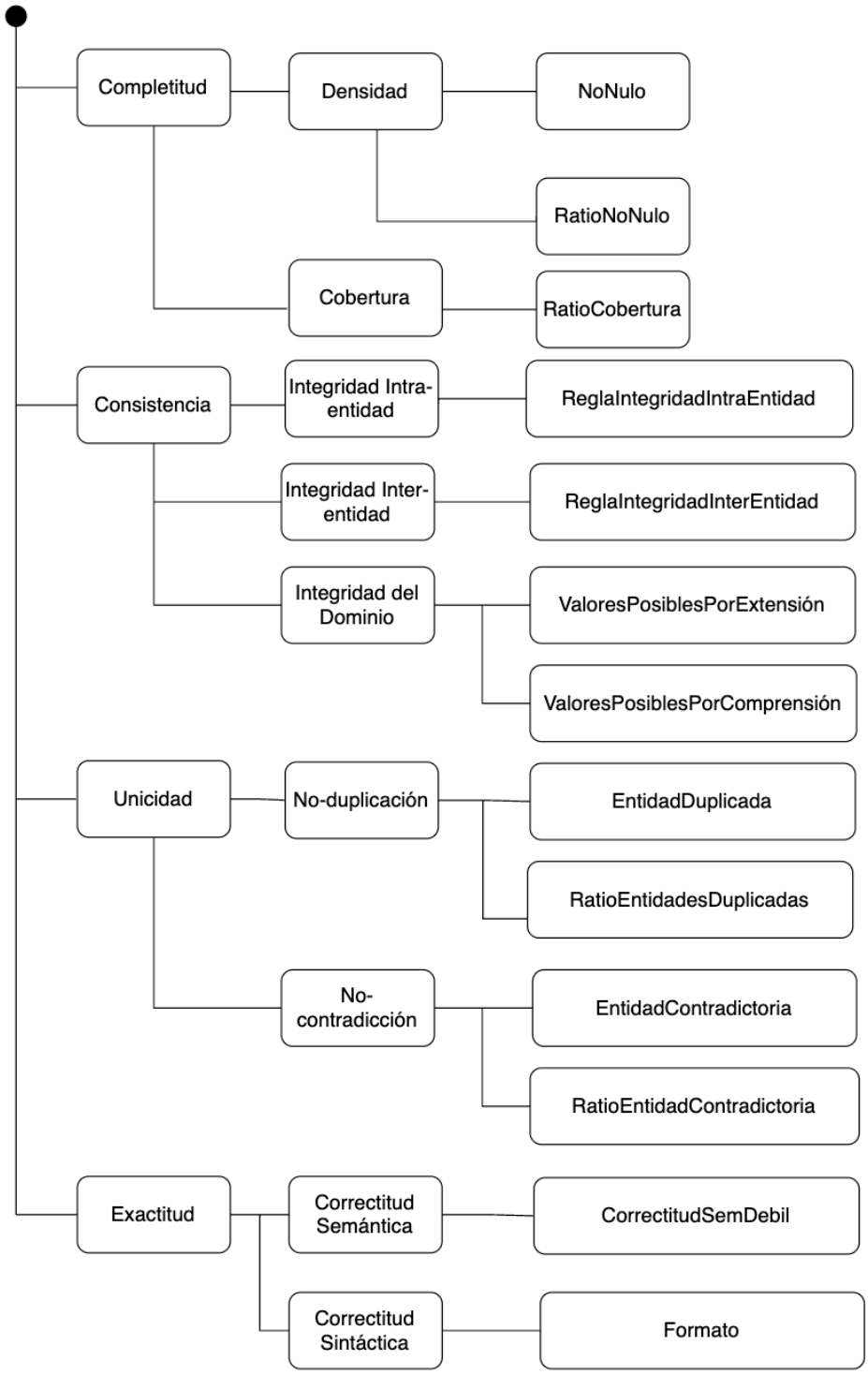

Figura 7.1: Modelo general del caso de uso

Como se puede ver en la Figura 7.1, las dimensiones Completitud, Unicidad, Consistencia y Exactitud son parte del modelo de referencia usado por la aplicación. Por lo tanto, cuando se creó el modelo y como se presenta en la Figura 7.2, se eligió la opción 'Usar Marco de Referencia', y se eliminaron las dimensiones, factores y métricas no necesarias. Es posible, además, agregar las dimensiones de Disponibilidad y Seguridad, con sus respectivos componentes, pero se decidió no incluirlas en el caso de uso. En el caso de Seguridad, se debe a que requiere el uso de parámetros sensibles como contraseñas y el sistema no ofrece ningún encriptado o mecanismo de seguridad al guardar esa información en la base. En el caso de Disponibilidad, se decidió no incluir esta dimensión porque los métodos que requiere no se adaptan bien al sistema. Para poder correrlos, habría que bloquear un núcleo de procesador mientras se chequea el servicio externo al sistema, y esto lo sobrecargaría mucho.

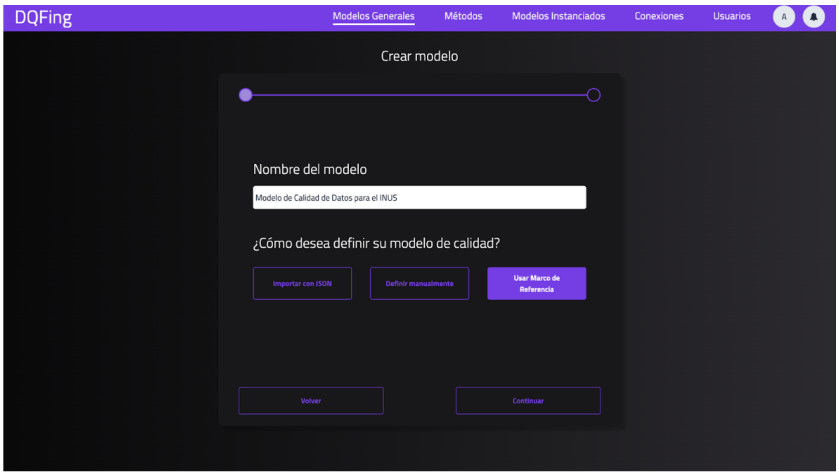

Figura 7.2: Formulario de creación modelo general paso 1

En la Figura 7.3 se observa el último paso del flujo de creación de modelo general. Aparte de presentar una vista completa de la estructura siendo creada, se pueden agregar y remover diferentes nodos del árbol.

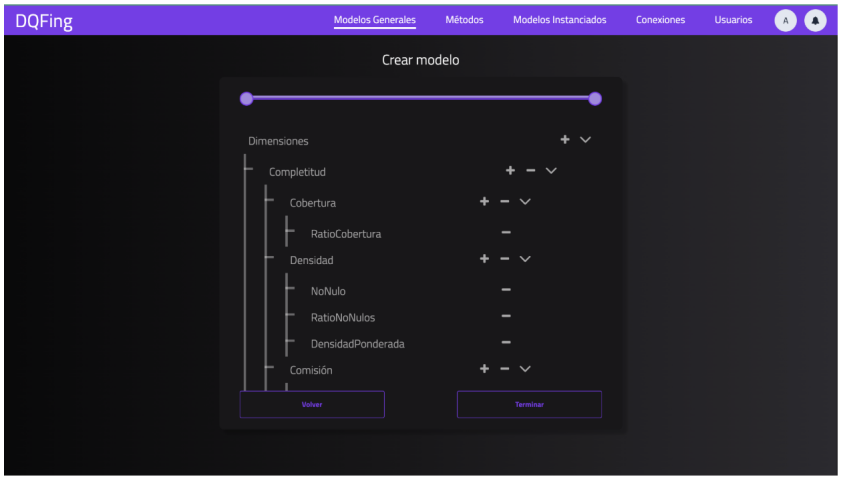

Figura 7.3: Formulario de creación modelo general paso 2

Una vez creado el modelo general, el mismo puede seguir siendo editado desde las tablas disponibles en la vista de cada modelo. Por ejemplo, en la vista de dimensiones que se presenta en la Figura 7.4, es posible agregar nuevas dimensiones, borrar o editar existentes, o seleccionar una dimensión para ver los factores que contiene. Las vistas de factores y métricas funcionan de forma análoga.

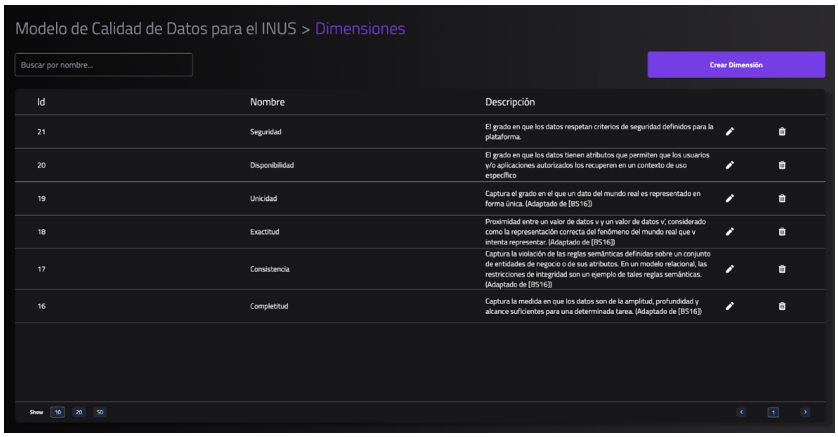

Figura 7.4: Tablas usadas para modificar modelos generales

En paralelo a la definición del Modelo General, es posible ir definiendo los métodos que se quieren usar para este modelo. Esto permite agilizar el proceso entre distintos usuarios, donde uno define el modelo general mientras otro puede ir trabajando en los métodos. Los métodos deberían ser lo más generales posibles ya que pueden ser usados en cualquiera de los modelos instanciados definidos dentro del

sistema. Son creados en un formulario de dos etapas, la primera de ellas se contempla en la Figura 7.5, donde se pueden definir los distintos parámetros y sujetos que se usarán dentro del método, mientras que en la segunda, presentada en la Figura 7.6, se puede escribir la lógica del método mediante código SQL o Python. Además de funciones nativas de ambos lenguajes, el sistema proporciona una batería de funciones adicionales propiamente descritas en el Anexo A, como se puede observar en el método escrito usando Python de la Figura 7.7.

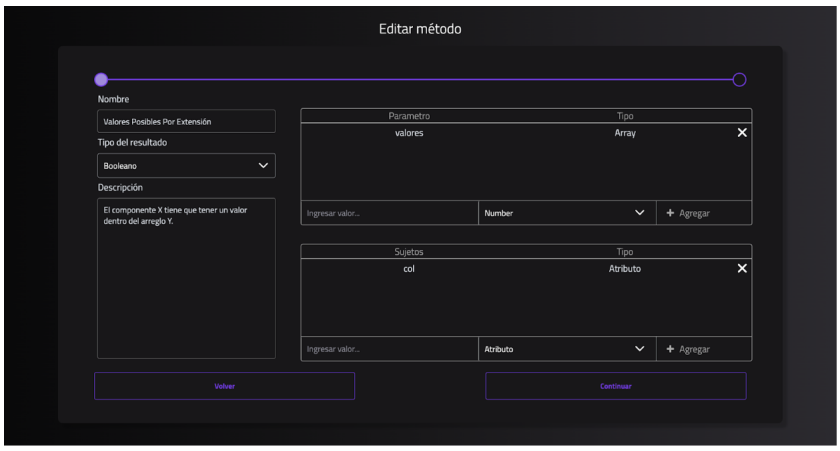

Figura 7.5: Formulario de creación métodos paso 1

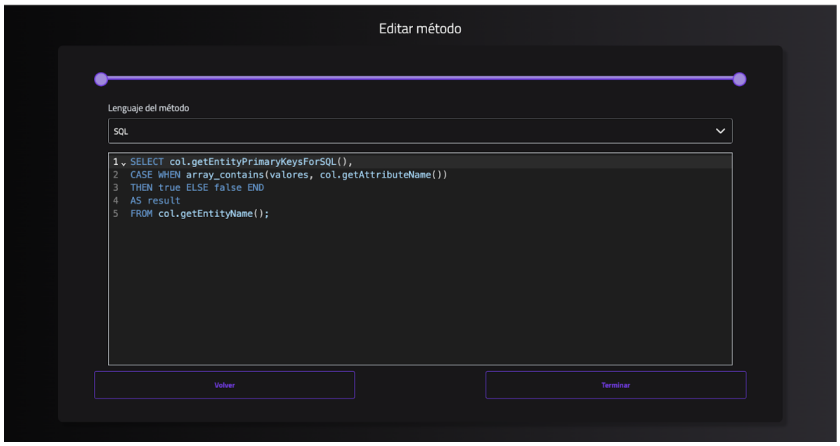

Figura 7.6: Formulario de creación métodos paso 2

Para este caso de uso, los métodos fueron definidos usando ambos SQL y Python, dependiendo de su complejidad. Métodos más simples como Formato, o NoNulo, fueron implementados directamente en SQL, pero métodos más complejos como

EntidadContradictoria, o CorrectitudSemDebil (que requería llamados a la API del DNIC para verificar cédulas), fueron implementados en Python. Esto se debe a que en Python es más fácil hacer verificaciones más complejas, mediante el uso de funciones o haciendo iterando sobre los datos.

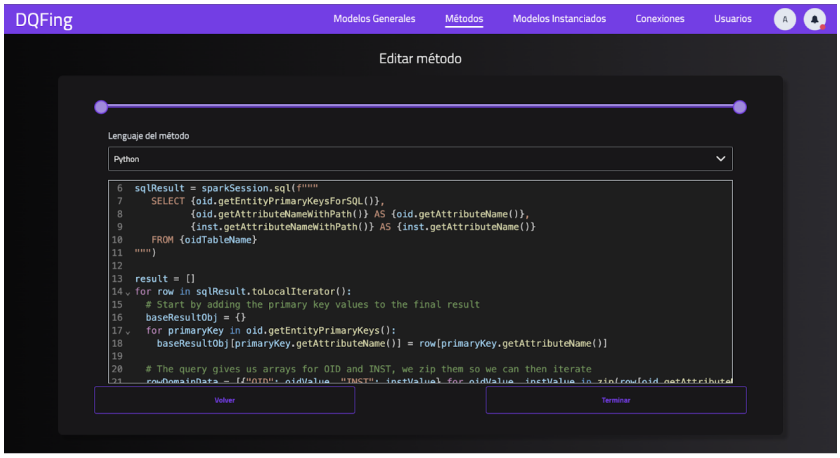

Figura 7.7: Ejemplo de metodo en Python

La complejidad de los métodos y de la estructura de los datos también afecta su re-usabilidad. Por ejemplo, cuando se tenían arreglos de datos como en el caso de la información de contacto (teléfonos, mails, etc.), se hacen métodos no reutilizables. En otros casos, más simples, con estructuras más accesibles, los métodos son más reutilizables. En total, para el caso de uso entero, se implementaron 22 métodos. En la Figura 7.8 se presenta la interfaz gráfica para la gestión de métodos, donde se pueden ver sus datos (p. ej. nombre, descripción) y realizar acciones sobre ellos (p. ej. eliminar).

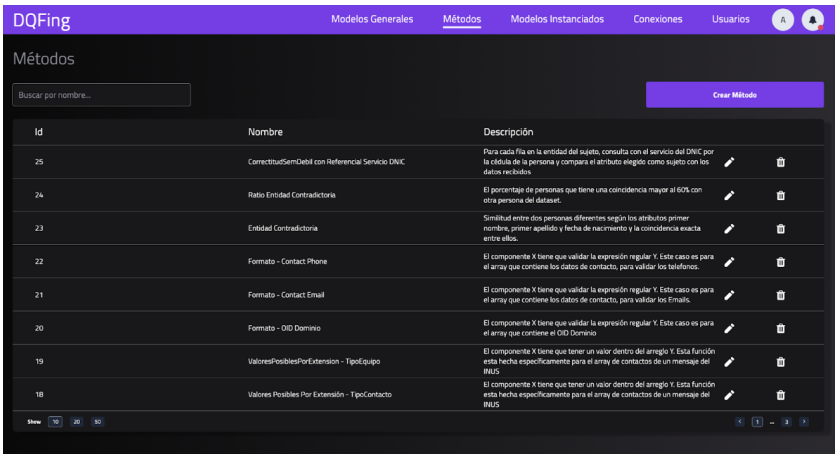

Figura 7.8: Tablas usadas para gestión de métodos

Para poder instanciar el modelo, es necesario crear una conexión a los datos dentro de la aplicación, en la pestaña conexiones. Esto también puede hacerse en paralelo a la definición de los métodos, y del modelo general. La aplicación permite crear conexiones de muchos tipos distintos, como bases de datos relacionales (PostgreSQL y MySQL), bases de datos documentales (MongoDB) y permite además subir archivos de datos (de tipo JSON o CSV). En este caso, los datos fueron representados mediantes dos archivos JSON, basados en la estructura de los mensajes con información de pacientes del INUS. Un JSON representa la base de datos del INUS en sí, mientras que el otro JSON, llamado 'Pacientes' representa una colección de mensajes recibidos mediante la API del INUS. Ambos JSON tienen la misma estructura. Una vez creados los datos dentro de la aplicación, la misma muestra el diagrama que se presenta en la Figura 7.9, usando el modelado de Coleccion, Entidad, Sub Entidad y Atributos del esquema de los datos.

Figura 7.9: Diagrama de la estructura de los datos del INUS

Este diagrama puede ser accedido desde la tabla de conexiones y es usado para instanciar el modelo más adelante.

Una vez que se tienen creados el modelo general, los métodos y la conexión a los datos, se puede crear un modelo instanciado. El modelo instanciado se compone de:

- Métricas instanciadas, donde se vinculan una métrica del modelo general, un método de evaluación y los sujetos y parámetros necesarios para la ejecución del método.
- *Jobs*, que se programan mediante expresiones *cron* y ejecutan todas las métricas del modelo instanciado de forma automática en segundo plano. También es posible ejecutar estos *jobs* de forma manual.

- Perfiles y reglas de evaluación, que permiten evaluar rápidamente si los datos se encuentran dentro de los rangos esperados. Se pueden definir diferentes perfiles a ser aplicados sobre las métricas instanciadas cada vez que se corre un job. Los perfiles son, sin embargo, opcionales y aunque se definan siempre es posible ver los resultados sin aplicar ningún perfil.

La Figura 7.10 muestra la interfaz destinada a la gestión de un modelo instanciado, desde la cual se accede a los distintos elementos previamente descritos. Cada componente cuenta con su propio listado, que incluye funcionalidades específicas para su creación, edición y eliminación.

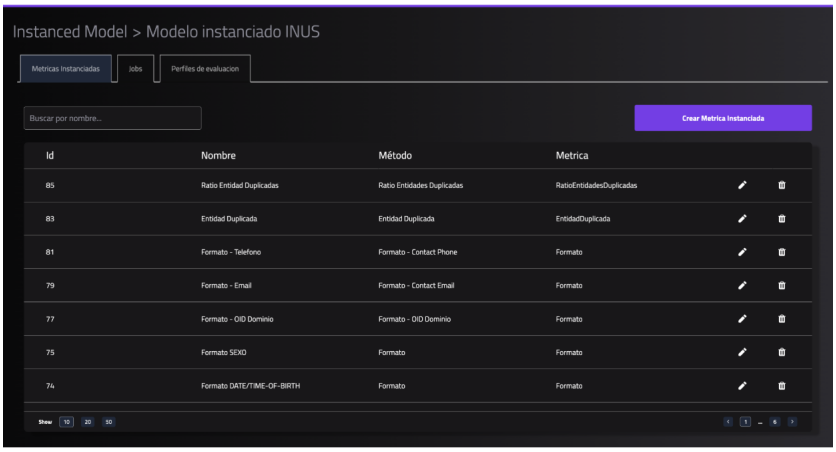

Figura 7.10: Modelo instanciado creado para el INUS

El formulario para instanciar una métrica, presentado en la Figura 7.11, permite al usuario seleccionar en primer lugar la métrica general que desea asignar, y a partir de esta selección, el sistema sugiere automáticamente los métodos compatibles. Una vez elegido el método, es posible definir los parámetros necesarios y seleccionar los sujetos utilizando los distintos diagramas correspondientes a las conexiones configuradas en el sistema. De este modo, un modelo instanciado puede incluir métricas aplicadas sobre diferentes fuentes de datos disponibles en la aplicación.

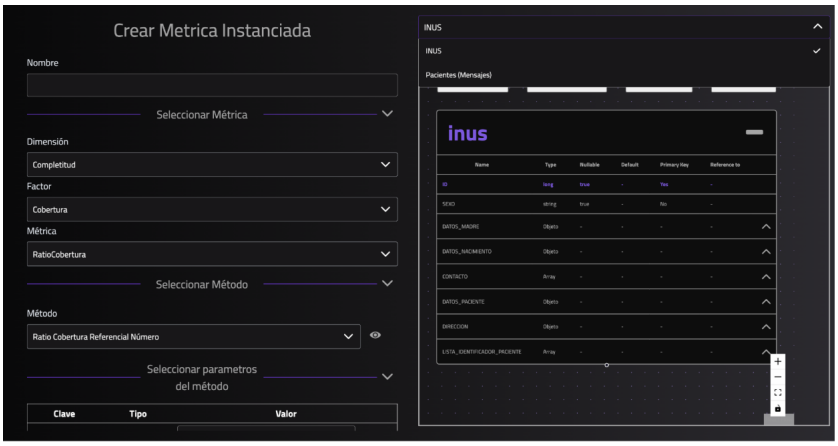

Figura 7.11: Formulario usado para la creación de métricas instanciadas

Para el caso del INUS, no se instancian todas las métricas del modelo. Como se mencionó previamente, no se implementaron los métodos para las dimensiones de Seguridad y Disponibilidad. Sin embargo, de los 8 requerimientos del modelo que contaban con métodos, se implementaron 6, dentro de un modelo instanciado llamado 'Modelo instanciado INUS'. El modelo cuenta con 64 métricas instanciadas. Para verificar que los métodos del sistema funcionan correctamente, se aprovechó el hecho de que los datos no son reales, y fueron generados por el equipo. Se generaron entonces datos con distintos resultados esperados para las distintas métricas instanciadas, para de esa forma verificar que los métodos retornaban resultados correctos.

Una vez definido el modelo instanciado, es posible definir *jobs* para ejecutarlo, o programar ejecuciones mediante un *cron job*, como se contempla en el formulario de la Figura 7.12. El usuario puede definir si quiere notificaciones para saber cuando termina la ejecución de cada *job*, o puede definir cuántas veces quiere que el *job* ejecute. Los *jobs* además pueden ser habilitados y deshabilitados, para definir si se quieren continuar o parar las ejecuciones automáticas.

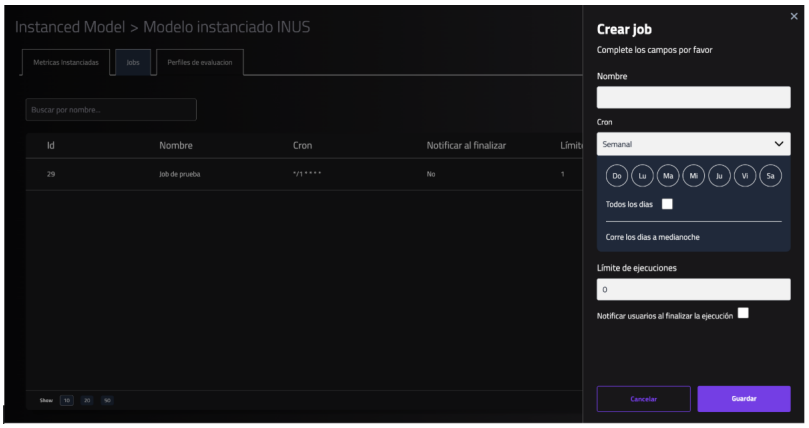

Figura 7.12: Formulario usado para la creación de *jobs*

Para cada *job*, se puede acceder a su lista de ejecuciones, y para cada ejecución, se puede acceder a sus resultados, como se ve en las figuras 7.13 y 7.14 respectivamente. Desde la página de resultados de una ejecución, es posible aplicar los distintos perfiles de evaluación definidos para esa ejecución.

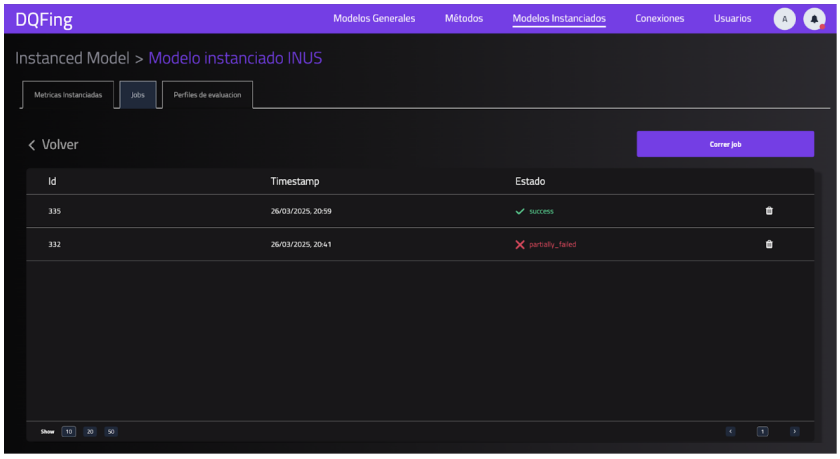

Figura 7.13: Vista de ejecuciones de un *job* para un modelo instanciado

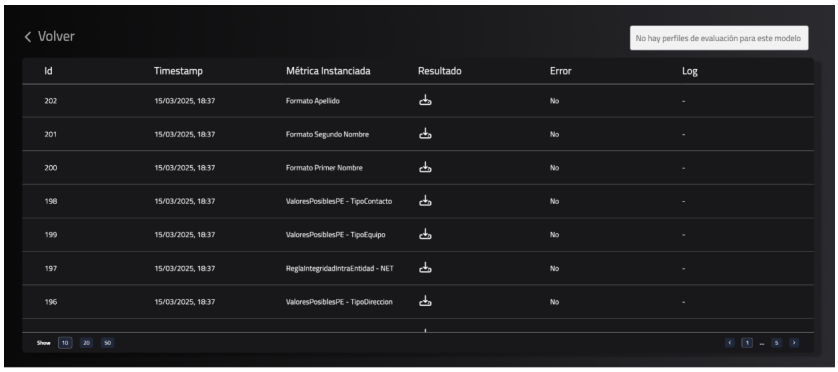

Figura 7.14: Detalle de ejecución de un *job*

Los perfiles de evaluación permiten definir distintos márgenes esperados para las ejecuciones de un modelo instanciado. Una vez definidos, los mismos son aplicados a cada ejecución, y pueden ser vistos desde la página de resultados de la ejecución que se muestra en la Figura 7.15. Los perfiles de evaluación no son aplicables a las métricas de granularidad instanciaAtributo o instanciaEntidad, ya que para estos casos el sistema ofrece archivos descargables para que el usuario analice los resultados. Sin embargo pueden ser usados en métodos más generales, y permiten hacer distintas comparaciones. Por ejemplo, en un caso donde no se quieren tener mails nulos en los datos, se podría definir una regla de evaluación que diga que la métrica instanciada RatioNoNulo, para el Email, debería ser igual a 100. Esto se podría ver más adelante en la vista de resultados, aplicando el perfil de evaluación donde se encuentra esa regla.

Figura 7.15: Detalle de ejecución de un *job*, con un perfil de evaluación

Dentro de esta vista, se cuenta además con un diagrama para ver rápidamente el porcentaje de reglas de evaluación que devuelven *true* o *false*, con la posibilidad de cambiar el perfil de evaluación aplicado.

7.3. Demo al equipo de Salud UY

Una vez completado el caso de uso, se realizó una demostración del sistema al equipo de Salud UY, responsables de la gestión del sistema de salud estatal. De este sistema, que incluye al INUS, es de donde se originaron los datos utilizados como base para el caso de estudio. En la reunión también participó la estudiante de maestría que diseñó el modelo de calidad implementado en el sistema.

En esta instancia se demostró un flujo común de uso de la aplicación, explicando el funcionamiento de cada componente de la misma. Se realizó además un video previo a la demo que contiene la misma información, para que quede como registro y documentación del proyecto.

La devolución recibida fue positiva, en particular se resaltaron los siguientes aspectos del sistema:

- El diseño de la aplicación con roles variados y con una interfaz amigable para usuarios técnicos y no técnicos.
- El diseño de la arquitectura, donde se remarcó la escalabilidad y buena distribución de la carga del sistema. En particular, tener el componente de procesamiento separado de los otros componentes del sistema, permite elevar los recursos a este componente de manera aislada, siendo el que se encarga del procesamiento más pesado de la aplicación.
- La posibilidad de tener varios niveles de negocio dentro de un mismo sistema, o sea, poder tener modelos de calidad para distintos rubros de una misma organización, separados dentro de una misma aplicación.
- El sistema ofrece una solución a una problemática que hasta el momento no había sido abordada. Anteriormente, las mediciones se realizaban de forma manual, en cambio, este sistema automatiza el proceso, centraliza los datos en un único lugar y permite contar con un historial completo de las mediciones realizadas, así como con perfiles de evaluación generados automáticamente.

Se plantearon además las siguientes mejoras o posibles problemáticas:

- Se mostró interés en ver qué tan bien la aplicación maneja realizar mediciones en sistemas con bases de datos muy grandes, para evaluar el rendimiento de la misma.
- Se planteó la posibilidad de agregar la funcionalidad de realizar copias de los métodos instanciados, para los casos donde se quiere instanciar dos modelos muy similares, pero con fuentes de datos distintas. Esto agilizaría la experiencia de usuario.

Capítulo 8

Conclusiones y Trabajo Futuro

El objetivo general del proyecto fue generar una aplicación capaz de correr métodos sobre distintas fuentes de datos, para brindar soporte al proceso de gestión de la calidad de datos. Para esto se realizaron una serie de reuniones con las tutoras del proyecto, en las cuales se definieron las principales características que debería tener la aplicación. En base a esto, se propuso una arquitectura orientada a maximizar la flexibilidad y la reutilización del sistema, ya sea mediante la creación de métodos que puedan aplicarse de forma indistinta a diferentes tipos de fuentes de datos, o mediante la separación del modelo de calidad en componentes —como el general y el instanciado— que permitan su reutilización.

También, se buscó crear una arquitectura con componentes separados y fáciles de reemplazar en caso de tener que adaptarse a otros sistemas o que en el futuro se identifiquen mejores tecnologías para solucionar el problema planteado a lo largo de este trabajo. Para favorecer la integración con otros sistemas, se incorporó también la posibilidad de importar modelos de calidad de datos en formato JSON. Esta funcionalidad permite tanto la interoperabilidad con el proyecto de Fiumarelli y Fornio (Fiumarelli y Fornio, 2020), como con cualquier otro proyecto que utilice los formatos admitidos por la aplicación.

Mientras que la arquitectura diseñada prioriza los requerimientos identificados como importantes, tiene ciertas limitaciones. Por ejemplo, la forma de modelar los datos se basa bastante en un modelo relacional, y requiere que los datos tengan un esquema. Esto no se aplica completamente a todos los casos de bases no relacionales, y presenta un problema abierto más grande y sin solución, que es la extracción de esquema de una base no relacional. Sin embargo, se aceptó esto como un compromiso para tener el beneficio de contar con métodos que puedan correr tanto en bases relacionales, como archivos JSON o CSV, o bases MongoDB.

Una vez definida la arquitectura, se continuó con el proceso de desarrollo de software, con una etapa de implementación utilizando distintas tecnologías, y se creó un prototipo funcional del sistema. Además, se llevaron a cabo etapas de verificación, que incluyeron tanto pruebas manuales como la implementación de pruebas automatizadas. También se realizó una etapa de validación, en la que se buscó aplicar un modelo de calidad real diseñado para el área de la salud, específicamente

para el componente INUS. Este proceso permitió realizar correcciones y mejoras en la aplicación, adaptándolo mejor a las necesidades reales y optimizando su usabilidad.

Mientras que se logró completar un proceso de desarrollo de software, desde el análisis de requerimientos hasta la validación, la aplicación aún tiene mucho espacio para mejorar y se identificaron posibles cambios y mejoras que no entraron dentro del marco de este proyecto. El trabajo a futuro identificado incluye:

- Inclusión de más fuentes de datos: hasta ahora el sistema permite conectarse a bases relacionales de PostgreSQL y MySQL, a archivos de tipo CSV o JSON, y a bases no relacionales MongoDB. A futuro, poder agregar otras fuentes tanto relacionales como no relacionales, o nuevos formatos de archivos (p. ej. formatos de *big data* Parquet u ORC), ayudaría a que el sistema sea aplicable a más casos de uso.

- Agregar una forma de acceder a las funciones del sistema sin necesitar del *frontend*. De esta forma sistemas externos que necesiten medir su calidad de datos podrían simplemente conectarse a la aplicación propuesta. Una forma de implementarlo sería con una API dedicada y un sistema de API keys que se asocien a una cuenta para limitar el acceso y los permisos.

- Sería conveniente implementar un sistema de versionado tanto para los modelos generales como para los instanciados. Actualmente, la aplicación no permite gestionar distintas versiones de los modelos generados, ni cuenta con un mecanismo para persistir y rastrear los cambios en ambos. Implementar este cambio permitiría al sistema reflejar de manera más precisa la naturaleza evolutiva del proceso de gestión de la calidad de datos, facilitando el control de modificaciones, la trazabilidad y la mejora continua.

- Realizar pruebas de rendimiento y proponer mejoras orientadas a sistemas de *big data*. Hasta el momento, la aplicación ha sido utilizada principalmente con bases de datos acotadas, y salvo por pruebas iniciales con grandes volúmenes de datos, no se ha explorado en profundidad cómo el sistema manejaría fuentes de datos masivas.

- Facilitar el proceso de desarrollo de nuevos métodos en el sistema con mejoras de experiencia de usuario, como por ejemplo mostrar la consulta ejecutada para las métricas instanciadas que usan métodos SQL, o agregar un sistema de *logging* para los métodos Python de forma de simplificar su verificación.

- Ampliar los formatos disponibles para la descarga de resultados de métricas instanciadas que devuelven datos por fila, además de Parquet. El objetivo de estos métodos es permitir que los usuarios analicen los resultados en sus propias aplicaciones. Para ello, una mejora sería ofrecer más opciones de formato, facilitando así que cada usuario elija el que mejor se adapte a sus necesidades o con el que esté más familiarizado.

- Permitir duplicar modelos instanciados, cambiando la base, para facilitar la creación de modelos similares.

- Implementar una validación más robusta de los resultados generados por los métodos. Actualmente, al ejecutarse un método, el sistema valida que el resultado sea consistente con el tipo esperado tanto por el método como por la métrica instanciada. Sin embargo, en los casos en que el método devuelve un resultado por fila, la validación se realiza únicamente sobre la primera fila, en lugar de verificar todos los resultados. Esta validación podría mejorarse procesando los resultados por lotes e, incluso, paralelizando dicho procesamiento para mejorar la eficiencia.

- Permitir la creación de parámetros secretos al definir un método, de modo que, al guardarse en la base de datos, se almacenen en forma encriptada o usando un algoritmo de *hash*. Esto permitiría aplicar métodos en contextos más amplios o con datos sensibles, como en pruebas de autenticación o autorización.

- Para la creación de métodos en Python, facilitar el uso de librerías externas a la aplicación. Hoy en día, las librerías disponibles se definen en un archivo, y las mismas son instaladas cuando se despliega la aplicación. Tener una forma dinámica de manejar esto haría que la aplicación sea más flexible para los usuarios finales, que no deberían pedir que se instalen las librerías cada vez.

- Poder llamar a métodos definidos dentro de la aplicación desde otros métodos definidos por la aplicación. De esta forma, los métodos serían capaces de referenciarse entre ellos, y se podría mantener una serie de funciones comunes re-utilizables a lo largo de las distintas implementaciones.

Referencias

Christopher Fox, A. L., y Redman, T. (1994). The notion of data and its quality dimensions. *Information Processing Management 30.1*.

Computación, I., y Agrimensura, F. d. I. (2020). Marco de trabajo para la gestión de la calidad de datos en gobierno digital [Manual de software informático]. https://www.gub.uy/agencia-gobierno-electronico-sociedad-informacion-conocimiento/sites/agencia-gobierno-electronico-sociedad-informacion-conocimiento/files/documentos/publicaciones/dq_framework_v1_6_compressed.pdf.

DataCleaner. (2019). *Reference documentation.* Descargado 2023, de https://datacleaner.github.io/docs/5.7.0/html/

Expectations, G. (2023). *Get started with great expectations.* Descargado 2023, de https://docs.greatexpectations.io/docs/

Fiumarelli, N., y Fornio, A. (2020). *Proyecto de grado: Plataforma para la gestión de calidad de datos en gobierno digital.* https://www.colibri.udelar.edu.uy/jspui/handle/20.500.12008/26453.

Foundation, T. A. S. (2018). *Apache griffin: Big data quality solution for batch and streaming.* Descargado 2023, de https://griffin.apache.org/docs/quickstart.html

GeoPandas Developers. (2022). *Geopandas documentation (stable version).* Descargado 2023, de https://geopandas.org/en/stable/docs.html

Inc., T. (2023). *Talend open studio user guide.* Descargado 2023, de https://help.talend.com/r/en-US/8.0/open-studio-user-guide

International, D. (2017). *Dama-dmbok: Data management body of knowledge: 2nd edition.* Technics Publications.

NumFOCUS, I. (2023). *User guide — pandas 2.1.1 documentation.* Descargado 2023, de https://pandas.pydata.org/docs/user_guide/index.html

Olson, J. E. (2003). *Data quality: the accuracy dimension.* Morgan Kaufmann.

OpenRefine. (2022). *Openrefine user manual.* Descargado 2023, de https://openrefine.org/docs

Paulheim, H. (2016, septiembre). Towards a definition of knowledge graphs. En *Joint proceedings of the posters and demos track of the 12th international conference on semantic systems (semantics 2016) and the 1st international workshop on semantic change & evolving semantics (success 2016)* (Vol. 1695). Leipzig, Germany: CEUR-WS.org. Descargado de http://ceur-ws.org/Vol-1695/paper4.pdf

Pidre, S. (2025). *Modelo de calidad de datos para la historia clínica electrónica nacional en un contexto interorganizacional.* (Reporte parcial de tesis de maestría en curso)

Rolland, A. (2022). *Mobydq: Automate data quality checks on data pipelines.* Descargado 2023, de https://ubisoft.github.io/mobydq/

Scannapieco, M., y Catarci, T. (2002). Data quality under a computer science perspective. *Archivi Computer 2, págs. 1-15.*.

Sourceforge. (s.f.). *Aggregate profile user guide version 5.5.10.* Descargado 2023, de https://sourceforge.net/p/dataquality/wiki/_discuss/thread/b06e41fa/8d5a/attachment/Profiler_User_Guide5.5.10.doc

Spark, A. (2023). *Pyspark overview.* Descargado 2023, de https://spark.apache.org/docs/latest/api/python/index.html

Tejada, Z. (2021). *Big data architectures.* Descargado 2024, de https://learn.microsoft.com/en-us/azure/architecture/databases/guide/big-data-architectures

Val Pushkarev, C. V., Henry Neumann, y Talburt, J. R. (2010). An overview of open source data quality tools. En *Proceedings of the 2010 international conference on information & knowledge engineering (ike 2010).* CSREA Press. (Source: DBLP)

Vohra, D. (2016). *Practical hadoop ecosystem.* New York, NY: Apress.

Wende, K. (2007). *A model for data governance-organising accountabilities for data quality management.* ACIS 2007 Proceedings, pag 80.

Anexo A: Instructivo de creación de métodos personalizados

A.1. Introducción teórica al sistema

A.1.1. Modelos de calidad de datos

Una de las partes principales del sistema se basa en la definición de modelos de calidad de datos. Estos se componen de una serie de elementos que permiten agrupar características favorables que deben poseer los datos. Los modelos se organizan en estructura de árbol, teniendo en primer lugar las dimensiones, que agrupan propiedades o características de calidad. Dentro de las dimensiones, se pueden definir los factores, que representan aspectos particulares de una dimensión de calidad (p. ej. correctitud sintáctica). Para cada factor se tiene a continuación una serie de métricas, que son instrumentos que definen la forma de medir un factor de calidad.

En el sistema, las métricas se ejecutan a través de métricas instanciadas, que permiten asociar las métricas con un método y una conexión de una fuente de datos. Se usan además para validar los parámetros que recibe el método y el resultado de la ejecución, utilizando la granularidad y el tipo de resultado esperado de su métrica.

Por una lado, la granularidad de una métrica indica a qué objeto de una o más fuentes de datos se aplica. A la hora de instanciar una métrica, se utiliza este dato para validar los objetos a los que se va a aplicar. Las granularidades soportadas por el sistema se pueden ver en la tabla de la Tabla A.1:

Granularidad	En Modelo Relacional
Instancia de atributo	Celda
Atributo	Columna
Conjunto de atributos	Conjunto de columnas
Instancia de entidad	Tupla
Entidad	Tabla
Conjunto de entidades	Conjunto de tablas
Colección	Base de datos
Conjunto de colecciones	Conjunto de bases de datos
Libre	-

Tabla A.1: Granularidades soportadas por el sistema

La granularidad libre permite instanciar métricas utilizando métodos que no dependen de una cantidad fija ni de un tipo específico de sujetos, e incluso admite métodos que no requieren sujetos en absoluto. En estos casos, el método se comporta de forma independiente a los datos disponibles.

Por otro lado, el tipo de resultado indica la salida esperada de una ejecución. Los tipos de resultado aceptados por el sistema son: porcentaje, real, entero, y verdadero o falso.

A.1.2. Esquemas de Conexiones

La ejecución de métodos en el sistema se realiza sobre conexiones a fuentes de datos creadas previamente. Al crear una nueva conexión, se extrae su esquema y se almacena utilizando una representación de colecciones, entidades, subentidades y atributos.

Colecciones

Una colección se corresponde con la fuente de datos a la que apunta una conexión. La información que se almacena sobre cada colección es su nombre, el de la base de datos, y el id de la conexión a la que pertenece.

Entidades

Las entidades se corresponden con tablas, en el caso de bases de datos relacionales, archivos, en el caso de que las fuentes sean CSV o JSON, y colecciones, en el caso de bases de datos MongoDB. La información que se almacena sobre cada entidad se puede ver en la Tabla A.2:

Dato	Descripción
Nombre	Nombre de la tabla, colección o archivo que le corresponda
Esquema	Nombre del esquema al que pertenece, en el caso de bases de datos que los usen, p.ej. PostgreSQL
Id de colección	Id de la colección a la que pertenece

Tabla A.2: Información almacenada en cada entidad

Subentidades

En el caso de bases de datos documentales y fuentes de datos con archivos JSON, la forma de representar los subdocumentos es a través de subentidades. Las subentidades son entidades que además pertenecen a otra entidad.

Se utiliza el término entidad raíz para entidades que no son subentidades, es decir que no tienen una entidad padre.

Se pueden usar entonces todas las funcionalidades de las entidades normales sobre estos objetos. Para facilitar el desarrollo y la implementación de métodos del lado del usuario, se limita la profundidad a la que llegan las subentidades del sistema a 3 niveles.

Una excepción de este modelado, es el caso del campo _id de MongoDB: este subdocumento es guardado como un atributo de tipo string que se considera automáticamente como una primary key.

Atributos

Los atributos representan las columnas de tablas y archivos CSV, y atributos de documentos de las bases de datos MongoDB y archivos JSON. La información que se almacena sobre cada atributo se puede ver en la Tabla A.3:

Dato	Descripción
Nombre	Nombre de la columna o atributo
Tipo	Tipo de la columna extraído directamente de la base de datos si esa información está disponible y usando PySpark en su defecto
Elemento	Si el tipo del atributo es array, indica el tipo de sus elementos, null sino
Valor por defecto	Valor por defecto de la columna si esa información está disponible en la base de datos, null sino
Nullable	Si la columna puede o no ser null, por defecto toma el valor true si no se puede extraer de la base de datos
Clave primaria	Si la columna es una clave primaria de la tabla a la que pertenece, por defecto toma el valor false si no se puede extraer de la base de datos. Nota: Solo se permite que las entidades raíz tengan claves primarias, es decir que si la entidad es una subentidad, este valor será siempre false
Id de clave foránea	Si el atributo es una clave foránea, indica el id del atributo que referencia, null sino
Id de entidad	Id de la entidad a la que pertenece

Tabla A.3: Información almacenada en cada atributo

Métodos

En este sistema, se entiende como un método al código que calcula el valor de alguna métrica. Generan entonces mediciones de calidad de datos. Para instanciar una métrica, se asocian:

- Una métrica, el objeto teórico que forma parte del modelo de calidad de datos,
- Un método, es decir el código a ejecutar,
- Una serie de parámetros de entrada y sujetos definidos por el método,
- Una fuente de datos.

Además, cada método permite especificar si devuelve un resultado único o un arreglo de resultados “por fila”. Esta última forma de devolver resultados puede aplicar tanto a un resultado por fila de una entidad como a cualquier otra serie de resultados.

Sujetos

Al instanciar una métrica, esta se aplica a uno o más objetos del esquema de una conexión. Estos objetos, que llamaremos sujetos, pueden ser colecciones, entidades

o atributos. Esto permite acceder a información general de la conexión a nivel de código a la hora de definir y ejecutar un método.

Parte de la configuración de un método indica los sujetos que éste debe recibir, especificando su tipo (atributo, entidad o colección) y el nombre por el que se identifica dentro del código. Esto se podría pensar como definir una variable, que puede ser invocada dentro del código junto con sus funciones asociadas. De esta forma se hace posible invocar métodos, definidos en el sistema, sobre los sujetos de la misma forma que una variable.

Existe una restricción que impide, en el momento de instanciar una métrica, elegir un método que no se corresponda en el tipo y cantidad de sujetos, con la granularidad de la métrica a instanciar. Los sujetos que acepta una métrica de cada granularidad se pueden ver en la tabla de la Table A.4:

Granularidad	Tipo de sujeto aceptado	Cantidad de sujetos aceptados
Instancia de atributo	Atributo	1
Atributo	Atributo	1
Conjunto de atributos	Atributo	1 o más
Instancia de entidad	Entidad	1
Entidad	Entidad	1
Conjunto de entidades	Entidad	1 o más
Colección	Colección	1
Conjunto de colecciones	Colección	1 o más
Libre	Todos	0 o más de cada tipo de sujeto

Tabla A.4: Tipo y cantidad de sujetos esperados por cada método según la granularidad de la métrica a la que se quiere asociar

Parámetros

Los parámetros que recibe un método pueden ser utilizados para pasar variables independientes del modelo de calidad y la conexión, con el objetivo de tener métodos más genéricos. El tipo de estas variables tiene que ser uno de los siguientes: number, string, boolean o arreglo de alguno de los anteriores. Para simplificar, el sistema no permite tipos más complejos como objetos, funciones, o arreglos de arreglos, por ejemplo.

Al instanciar una métrica el sistema valida que los parámetros instanciados coincidan en tipo con el especificado en el método.

A.2. Advertencia

Antes de pasar a los detalles de implementación de métodos personalizados, advertimos que todos los métodos del sistema son ejecutados sobre bases de datos reales. Es necesario ser prudente a la hora de crear nuevos ya que el sistema no restringe de ninguna forma el código ejecutado. Esto quiere decir que los usuarios del sistema pueden llegar a realizar cualquier tipo de operación sobre la base de datos de una conexión, y se debe velar por la integridad de los datos accedidos.

Una forma simple de mitigar estos peligros es utilizar usuarios con permisos restringidos al momento de crear las conexiones de datos (por ejemplo solo de lectura).

De todas formas, recomendamos ser precavidos a la hora de escribir el código de un nuevo método, y es por eso que la creación de métodos se restringe solo a usuarios técnicos del sistema.

A.3. Creación de métodos

Para crear métodos, es necesario proporcionar ciertos atributos inicialmente, por ejemplo un nombre, una descripción, un tipo de resultado, la cantidad de estos y los previamente mencionados sujetos y parámetros. Además de estos, se debe introducir el código del método a ejecutar, eligiendo entre dos lenguajes posibles, Python y SQL. Ambos lenguajes tienen sus ventajas y limitaciones por caso. Se presenta a continuación distintas funcionalidades y capacidades del sistema, que funcionan tanto para ambos lenguajes como para alguno de los dos en específico.

A.3.1. Funcionalidades compartidas

Tanto en SQL como en Python, los métodos admiten el uso de los parámetros como variables. Además, para el uso de los sujetos, los mismos cuentan con funciones para obtener información sobre el sujeto mismo o su conexión. Estas funciones pueden ser específicas para cierto tipo de sujeto, devolviendo un valor nulo en caso de que no aplique. Las funciones disponibles son las presentadas en la Table A.5:

Función	Descripción	Tipo de sujeto al que aplica
getConnection()	Devuelve el objeto de la conexión a la que el *sujeto* está relacionado.	Todos
getCollectionName()	Devuelve el nombre de la colección a la que pertenece el *sujeto*	Todos
getCollectionEntities()	Devuelve los objetos de las entidades que corresponden a la colección a la que pertenece el *sujeto*	Todos
getEntitySchema()	Devuelve el nombre del esquema al que pertenece el *sujeto*, devuelve un valor nulo en caso de que la entidad no tenga un esquema	Atributo y Entidad
getEntityName()	Devuelve el nombre de la entidad del *sujeto*, devuelve un valor nulo en caso de que la entidad no tenga un esquema.	Atributo y Entidad
getTopLevelEntityName()	En el caso donde la entidad sea raíz, se comporta igual que la función anterior. En caso de que el sujeto sea una subentidad, devuelve en su lugar el nombre de la entidad raíz.	Atributo y Entidad
isTopLevelEntity()	Devuelve *True* si la entidad del sujeto es raíz y *False* si no.	Atributo y Entidad
getEntityPrimaryKeys()	Devuelve los objetos de los atributos que conforman la clave primaria de la entidad raíz del *sujeto*. Nota: Esta función devuelve un error en caso de que la entidad sobre la que se llame no tenga claves primarias.	Atributo y Entidad
getEntityPrimaryKeysForSQL(prefix=None)	Devuelve un string con los nombres de las claves primarias de la entidad raíz del *sujeto* separados por comas. En caso de pasar un prefijo, este se posiciona antes de cada clave, separado por un punto. Se espera que el prefijo se pase a la función como un string, es decir entre comillas que pueden ser simples o dobles. Nota: Esta función devuelve un error en caso de que la entidad raíz sobre la que se llame no tenga claves primarias.	Atributo y Entidad
getAttributeName()	Devuelve el nombre del atributo del *sujeto*.	Atributo
getAttributeNameWithPath()	En el caso donde el atributo pertenezca a una entidad raíz, se comporta igual que la función anterior. Si el atributo pertenece a una subentidad, devuelve los nombres de las subentidades en orden descendente separados por puntos, sin incluír el nombre del atributo raíz, concatenando al final el nombre del atributo.	Atributo

Tabla A.5: Funciones disponibles para métodos escritos en SQL o Python

Ejemplo

Se toma como ejemplo la entidad Mensaje presentada en el fragmento de Código A.1, basada en un archivo JSON donde un elemento tiene el siguiente formato, siendo *id* la clave primaria de la entidad.

```
{
    "id": 12345,
    "data": {
        "content": {
            "text": "This is a message"
        },
        "timestamp": "1738971005"
    }
}

```

Código A.1: Estructura JSON de la entidad Mensaje

Por un lado, si al instanciar una métrica se elige como sujeto *entity* (nombre elegido de parte del desarrollador para referenciar a una entidad) a la entidad *content* se obtendrían los resultados presentados en el Código A.2.

```
entity.getTopLevelEntityName()
# "Mensaje"


entity.getEntityPrimaryKeysForSQL()
# "id"


```

Código A.2: Ejemplo usando funciones sobre un sujeto de tipo entidad

Por otro lado, instanciando una métrica con un sujeto de tipo atributo (p. ej. nombrado *attr*) asignado al atributo *text*, se obtendrían resultados como los contemplados en el Código A.3.

```
  attr.getAttributeNameWithPath()
  # "data.content.text"

```

Código A.3: Ejemplo usando funciones sobre un sujeto de tipo atributo

Además, para las funciones como getEntityPrimaryKeys que devuelven objetos, estos pueden ser tratados de la misma forma que un sujeto. Por ejemplo para la función getEntityPrimaryKeys, se pueden invocar sobre los atributos resultantes las funciones de los sujetos, como getAttributeName.

A.3.2. Python

Funciones auxiliares

Al escribir un método en Python, el programador tiene disponible, además de los parámetros y los sujetos, el objeto sparkSession. Este objeto tiene funcionalidades de PySpark que se pueden usar libremente dentro del método, y también puede ser

pasado a las funciones auxiliares provistas por la aplicación. Las funciones para el manejo de las sparkSession se presentan en la Tabla A.6:

Función	Descripción
readTable(sparkSession, connection, tableName, tableSchema?)	Devuelve un DataFrame de PySpark, con los datos de la tabla especificada (*tableName*) de la conexión pasada en el segundo argumento. Es posible además pasar un esquema en caso de que sea necesario.
createTempView(sparkSession, connection, tableName, tableSchema?, viewName?=tableName)	Crea una vista temporal local de la tabla especificada de la conexión pasada dentro de la sesión de spark. Esto no carga la tabla entera en memoria, pero permite realizar consultas SQL posteriormente a la vista creada. Al igual que en la función readTable, es posible pasar un esquema en caso de ser necesario, también se puede proveer un nombre para la nueva vista, que en caso de no pasarse toma como valor el nombre de la tabla por defecto. Las consultas que se hagan a la vista temporal creada deberán usar el nombre de la vista como nombre de la tabla.

Tabla A.6: Funciones para el manejo de las sparkSession

Estas funciones no proveen funcionalidades específicas relacionadas a la aplicación más allá de utilizar los datos de la conexión, su objetivo es facilitar acciones comunes y estandarizar la obtención de datos de la base de datos de la aplicación.

Cómo escribir un método

Un método debe, en algún momento de su ejecución, asignarle su resultado a la variable result, cuyo valor inicial es el valor None. Esta variable es reconocida automáticamente por el sistema y es donde se espera el resultado que será luego procesado y almacenado.

La ejecución de una métrica instanciada, al estar asociada tanto a un método como a una métrica, chequea que el resultado final sea del tipo especificado por la métrica. En caso de que esto no coincida, se marca la ejecución como fallida y se almacena un mensaje de error que incluye el resultado obtenido y el tipo esperado.

En caso de que el resultado sea “por fila”, se chequea también que el mismo sea un arreglo de objetos JSON o strings con formato JSON.

El sistema no chequea la validez del código de un método en ningún momento. Sin embargo, cualquier error en tiempo de ejecución causa que la ejecución de la métrica instanciada se marque como fallida y se almacene el mensaje de error, para luego ser mostrado en el portal web.

Ejemplo

Se toma como ejemplo el método que se presenta en el Código A.4, escrito usando Python, que dado un sujeto col de tipo atributo y un parámetro de tipo arreglo validValues, obtiene el porcentaje de las filas que tienen un valor válido (respecto a validValues) en la columna del sujeto del método.

```
1 tableName = col.getTopLevelEntityName()
2 createTempView(sparkSession, col.getConnection(), tableName, col.
      getEntitySchema())
```

```
3
4
5  totalData = sparkSession.sql(f"SELECT count(*) as res FROM {
       tableName}")
6  semanticallyCorrectData = sparkSession.sql(f"""
7     SELECT count(*) as res
8     FROM {tableName}
9     WHERE {col.getAttributeNameWithPath()} IN ({', '.join([f'"{value
       }"' for value in validValues])})
10 """)
11
12
13 result = (semanticallyCorrectData.first().res) / (totalData.first()
       .res)
```

Código A.4: Ejemplo de método en Python

Se puede ver cómo en primer lugar, se obtiene el nombre de la entidad del sujeto utilizando una de las funciones que estos tienen disponibles, para luego usar la función auxiliar createTempView, usando la sesión de spark, la conexión a la que está relacionada el sujeto, el nombre de la entidad y el del esquema.

Una vez creada la vista temporal, la sesión de spark es capaz de realizar consultas SQL a esta para obtener el total de elementos en la entidad y la cantidad de elementos que cumplen la condición de que el atributo del sujeto sea semánticamente correcto.

Finalmente, se calcula el resultado del método a partir de los valores obtenidos en las consultas y, para devolverlo, se le asigna ese valor a la variable result.

Utilización de librerías

El sistema permite utilizar tanto librerías incluidas en Python como librerías externas en los métodos, con el objetivo de proveer la mayor cantidad de posibilidades al programador. Las librerías de terceros incluídas en el proyecto son:

- Requests, que permite realizar consultas http, en caso de necesitar acceder a APIs de terceros.
    - En este caso es trabajo de la persona que escriba el método realizar los pasos necesarios para obtener autorización para usar la API a la que se le están realizando consultas. Se podrían utilizar eventualmente parámetros para cargar por ejemplo una API key o un usuario y contraseña, teniendo en cuenta de todas formas que estos se guardan en texto plano en la base de datos del sistema.
- Numpy, que permite realizar cálculos matemáticos eficientemente trabajando con grandes cantidades de datos.
- Pandas, que permite analizar y procesar datos utilizando data frames, con una API compatible con la de PySpark.

Ejemplo

En el Código A.5 se puede observar otro ejemplo de método, que utiliza la librería requests para realizar una consulta http a una API externa al sistema (The Star Wars API), donde obtiene la lista de películas de Star Wars. Una vez que se obtiene la respuesta a la consulta, se procesa para extraer los nombres de las películas y usarlas como los valores válidos a chequear posteriormente en el atributo elegido como sujeto del método.

```
import requests


response = requests.get('https://swapi.dev/api/films')


if response.status_code != 200:
   raise Exception('Failed to load data')


data = response.json()


validValues = [f'"{film.get("title")}"' for film in data.get("
    results")]


tableName = col.getTopLevelEntityName()
createTempView(sparkSession, col.getConnection(), tableName, col.
    getEntitySchema())


totalData = sparkSession.sql(f"SELECT count(*) as res FROM {
    tableName}")


semanticallyCorrectData = sparkSession.sql(f"""
   SELECT count(*) as res
   FROM {tableName}
   WHERE {col.getAttributeNameWithPath()} IN ({', '.join(
    validValues)})
""")


result = (semanticallyCorrectData.first().res) / (totalData.first()
    .res)
```

Código A.5: Ejemplo de método utilizando librerías adicionales

Se observa cómo se importa la librería dentro del método y luego se utiliza normalmente, pudiendo incluso realizar operaciones asíncronas y levantar excepciones cuando sea necesario. En este último caso, el mensaje de error que se le pasa a la excepción es el que se guarda asociado a la ejecución fallida de la métrica instanciada.

Nota Adicional

Durante el desarrollo de la aplicación y de distintos métodos de prueba, se observaron distintos errores en código python que normalmente sería válido. Esto se debe a la forma en la que se ejecutan los métodos, utilizando la función exec de Python. Los errores se relacionan a problemas de alcance de las variables declaradas dentro de un método, ya sean de tipos primitivos, funciones o incluso objetos importados como se mostró en la sección anterior.

Es por esto que se recomienda evitar declarar funciones dentro de un método y, en caso de ser necesario, que se intente evitar utilizar dentro de una función variables declaradas fuera de ella. Se debería entonces, por ejemplo:

- Importar una librería dentro de una función en lugar de al principio de un método, si es que solo se precisa dentro de la función.
    - En este caso la solución sería importar la librería dentro de la función que la requiere.
- Declarar dos funciones en la raíz de un método para usar una dentro de la otra
    - Aquí la solución podría ser declarar una función dentro de la otra.

A.3.3. SQL

Utilización de parámetros y sujetos

Como se mencionó anteriormente, los métodos SQL pueden usar los parámetros y sujetos que tengan definidos. Durante la ejecución de una métrica instanciada, se realiza un preprocesamiento del código del método buscando las ocurrencias de los parámetros y las funciones usadas sobre los sujetos para reemplazarlos por los valores que les corresponden según la configuración de la métrica instanciada.

Como los resultados se obtienen reemplazando manualmente las ocurrencias por valores, en el caso de los sujetos el sistema solo reemplaza las llamadas a las funciones que devuelven un string. Esto quiere decir que no se tienen en cuenta las funciones que devuelven objetos, como getConnection, getCollectionEntities y getEntityPrimaryKeys.

Además, en los casos donde no se encuentre un resultado para la función llamada, por ejemplo si el sujeto es de tipo entidad y se utiliza la función getAttributeName, el sistema marcará la ejecución como fallida con un mensaje de error descriptivo.

Durante el preprocesamiento del código, al reemplazar las ocurrencias de la función getEntityName sobre entidades raíz, el sistema se encarga de llamar la función auxiliar createTempView, vista previamente en la sección de métodos en Python. El objetivo de esto es crear las vistas temporales necesarias para poder ejecutar la consulta.

Cómo escribir un método

Los métodos SQL consisten en consultas que se ejecutan utilizando la función sql de PySpark. Debido a esto, el dialecto utilizado en las queries es SparkSQL (Documentación oficial), que permite, entre otras cosas, realizar consultas a bases de datos no relacionales.

El sistema espera que la consulta de un método SQL devuelva un único valor en una columna llamada result, excepto que el método especifique que devuelve resultados "por fila". En este casos el resultado debe tener:

- una columna result que contiene el resultado del método para cada fila, y
- por lo menos una columna más que sirva para identificar a la fila dentro del resultado total

Ejemplo 1

Se toma como ejemplo el método presentado en el Código A.6, escrito usando SQL, que dado un sujeto *col* de tipo atributo, obtiene, para cada fila de la entidad que le corresponde, si el atributo es no nulo.

```
SELECT col.getEntityPrimaryKeysForSQL(),
       CASE WHEN col.getAttributeNameWithPath() IS NULL
       THEN False ELSE True END
       AS result
FROM col.getTopLevelEntityName();
```

Código A.6: Ejemplo de método para RatioNoNulo en SQL

Se observa cómo en primer lugar, se obtienen los nombres de las claves primarias de la entidad del sujeto utilizando una de las funciones que estos tienen disponibles. A continuación se define la columna result utilizando la sintaxis habitual de un CASE en SQL y se utiliza otra de las funciones disponibles en el sujeto para obtener el nombre de su entidad relacionada.

Ejemplo 2

En el ejemplo expuesto en el Código A.7 se tienen dos sujetos *colA* y *colB*, ambos de tipo atributo, y un parámetro *givenValues* de tipo arreglo. Se usan los nombres de los atributos de ambos sujetos en las condiciones de un *CASE*, el primero para chequear que su valor este contenido en el arreglo y el segundo para chequear si no es nulo.

```
SELECT (
       COUNT(
           CASE
               WHEN colA.getAttributeNameWithPath() IN (givenValues
    )
               AND colB.getAttributeNameWithPath() IS NOT NULL
           THEN 1 END
       ) * 100.0 / COUNT(*)
   ) AS result
```

```
FROM colA.getTopLevelEntityName();
```

Código A.7: Ejemplo de método en SQL

Este método podría aplicarse por ejemplo a una métrica de ratio de integridad intra-entidad, donde se busca que si un atributo tiene un valor especificado previamente, otro atributo de la misma entidad debería tener un valor no nulo. Esta métrica, al ser de granularidad conjunto de atributos, admite más de un sujeto de tipo atributo.

Se observa además un detalle de la implementación del preprocesamiento del código SQL por parte de la aplicación, que en caso de recibir un parámetro de tipo arreglo, lo formatea para que pueda ser utilizado en una consulta. Por ejemplo, un resultado posible de la etapa de preprocesamiento, podría ser el presentado en el Código A.8.

```
SELECT (
       COUNT(
           CASE
               WHEN type IN ('mysql', 'mongodb', 'postgres')
               AND host IS NOT NULL
           THEN 1 END
       ) * 100.0 / COUNT(*)
   ) AS result
FROM connections;
```

Código A.8: Ejemplo de método en SQL preprocesado

En este caso, los nombres de los atributos de los sujetos colA y colB son 'type' y 'host' respectivamente, y el parámetro givenValues es un arreglo que contiene los valores 'mysql', 'mongodb' y 'postgres'.

Aplicación a bases de datos no relacionales

Se mencionó en el apartado anterior que SparkSQL permite hacer consultas a bases de datos no relacionales como si fueran relacionales. Combinando esto con las funciones de los sujetos, se llega a que, en muchos casos, los métodos pueden ser aplicados independientemente de que la fuente sea relacional.

Además, el dialecto tiene disponibles funciones específicas para manejar datos como objetos JSON o arreglos, que pueden facilitar la lógica de una consulta o el procesamiento a realizar. Esto no solo aplica a bases de datos MongoDB, sino que es útil también cuando se trabaja con conexiones basadas en archivos JSON o cuando una tabla de una base de datos relacional tiene una columna de tipo JSON.

Ejemplo

En la consulta presentada en el Código A.9 se tiene un sujeto *arrCol*, que se espera que sea un atributo de tipo arreglo de *string*. Utilizando las funciones *LATERAL VIEW* y *EXPLODE* de SparkSQL, se obtiene una tabla intermedia partiendo desde la entidad de *arrCol*. Esta tabla tiene una fila para cada elemento del arreglo del atributo, por cada fila de la tabla original, en una nueva columna que en este caso se nombra *columnFromArray*.

```
WITH explodedTable AS (
 SELECT arrCol.getEntityPrimaryKeysForSQL(),
        columnFromArray
 FROM arrCol.getTopLevelEntityName()
 LATERAL VIEW EXPLODE(arrCol.getAttributeNameWithPath()) AS
    columnFromArray
)
SELECT
 (
   COUNT(arrCol.getEntityPrimaryKeysForSQL()) * 100
 ) /
 (
   SELECT COUNT(*) FROM explodedTable
 ) AS result
FROM explodedTable
WHERE regexp_like(columnFromArray, regexFormat)
```

Código A.9: Ejemplo de método en SQL utilizando funciones de SparkSQL

El método usa la nueva tabla *explodedTable* para ejecutar la función *regexp_like*, también provista por SparkSQL, sobre el atributo del sujeto, que toma un *string* y una expresión regular, que en este caso utiliza el parámetro *regexFormat*. Finalmente, se calcula el porcentaje de elementos de todos los arreglos de la tabla que cumplen el formato especificado en el parámetro.

De la misma forma que en caso de que un parámetro sea un arreglo se formatea de forma específica durante el preprocesamiento de la consulta, en caso de ser un *string*, se ponen comillas simples alrededor del resultado, como se ve en el Código A.10.

```
WITH explodedTable AS (
 SELECT userId1, userId2,
        email
 FROM userEmails
 LATERAL VIEW EXPLODE(emails) AS email
)
SELECT
 (
   COUNT(userId1, userId2) * 100
 ) /
 (
   SELECT COUNT(*) FROM explodedTable
 ) AS result
FROM explodedTable
WHERE regexp_like(email,    ^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-
    zA-Z]{2,} $  )
```

Código A.10: Ejemplo de método en SQL utilizando funciones de SparkSQL

En este caso se estaría aplicando el método presentado en el Código A.10 a una tabla de usuarios con una clave primaria compuesta por dos *IDs* y una columna *emails* que tiene un arreglo de correos del usuario. La expresión regular usada sirve para reconocer direcciones válidas de correo electrónico.

Extracción del resultado

Después de ejecutar la consulta, el sistema procede a extraer el resultado, usando la granularidad de la métrica instaciada para saber si se espera un único valor o una tabla con un resultado por cada fila de la entidad original.

A.3.4. Validación de resultados

Una vez que se ejecuta el método de una métrica instanciada, se utilizan las propiedades de la métrica que se instanció para validar que el tipo del resultado es el esperado. Existen dos casos posibles para este chequeo:

- Si el método devuelve resultados "por fila", para evitar una carga excesiva en esta etapa, se toma únicamente la primera fila del resultado y se valida que haya:
  - una columna result, que se chequea además que sea del tipo de resultado esperado por la métrica, y
  - alguna otra columna, para ser usada como clave primaria dentro del resultado del método, vale aclarar que no se chequea la unicidad de estas claves
- En el caso contrario, se realiza únicamente la validación del tipo del resultado.

A.3.5. Evaluación de resultados

Además de validar los resultados de la ejecución de métodos, la aplicación tiene un sistema de evaluación de estos a través de perfiles de evaluación. Estos perfiles están atados a los modelos instanciados y sirven como conjuntos de reglas de evaluación. No hay un límite en la cantidad de perfiles de evaluación que puede tener un modelo instanciado.

Una regla de evaluación está relacionada con una métrica instanciada y debe tener definidas una operación y un valor a comparar, que debe coincidir con el tipo de resultado especificado en la métrica. Las posibles operaciones para las reglas son:

- Mayor que
- Mayor o igual que
- Igual a
- No igual a
- Menor o igual que
- Menor que

Anexo B: Formatos de JSONs importables para cargar modelos generales

La aplicación desarrollada permite la creación de modelos de calidad de datos personalizados mediante la carga de archivos JSON con una estructura definida. En el Código B.1 se presenta el primero de los formatos aceptados. El mismo es una versión adaptada del formato exportable del proyecto de 2020, el cual es visible en el Código B.2.

```
{
    "name": "Nombre modelo general",
    "dimensions": [{
        "name": "Nombre dimension",
        "description": "Descripcion de la dimension",
        "factors": [{
            "name": "Nombre factor",
            "description": "Descripcion del factor",
            "metrics": [{
                "name": "Nombre metrica",
                "semantic": "Descripcion de la metrica",
                "granularity": 1 // Debe ser mapeado a uno de los valores del GranularityEnum de nuestra aplicacion
                "result_type": 1 // Debe ser mapeado a uno de los valores del ResultTypeEnum de nuestra aplicacion
            }]
        }]
    }]
}
```

Código B.1: Estructura JSON aceptada por la aplicación para la creación de modelo

```
{
    "dimensions": [{
        "name": "Nombre dimension",
        "alternative_names": "Nombres alternativos (este campo es ignorado)",
        "definition": "Descripcion de la dimension",
        "factors": [{
            "name": "Nombre factor",
```

```
            "alternative_names": "Nombres alternativos (este campo es ignorado)",
            "definition": "Descripcion del factor",
            "metrics": [{
                "name": "Nombre metrica",
                "semantics": "Descripcion de la metrica",
                "granularity": 1 // Debe ser mapeado a uno de los valores del GranularityEnum de nuestra aplicacion
                "result_type": 1 // Debe ser mapeado a uno de los valores del ResultTypeEnum de nuestra aplicacion
                "sugested_name": "Nombre sugerido" // Este campo es ignorado
                "properties": [] // Este campo es ignorado
            }]
        }]
    }]
}
```

Código B.2: Estructura JSON aceptada por la aplicación para la creación de modelo

Como se puede observar en los fragmentos de Código B.1 y B.2, la propiedad *granularity* acepta valores enteros que se especifican más detalladamente en el Código B.3.

```
enum GranularityEnum {
  instanciaAtributo = 1,
  atributo = 2,
  conjuntoAtributos = 3,
  instanciaEntidad = 4,
  entidad = 5,
  conjuntoEntidades = 6,
  coleccion = 7,
  conjuntoColecciones = 8,
}
```

Código B.3: Valores posibles para el atributo granularity

Por otro lado, de igual manera para el atributo *result_type*, se tiene un conjunto de valores enteros posibles que se pueden observar en detalle en el Código B.4.

```
enum ResultTypeEnum {
  percentage = 1,
  boolean = 2,
  real = 3,
  integer = 4,
}
```

Código B.4: Valores posibles para el atributo result_type

Anexo C: Historias de usuarios de la aplicación

1. **Inicio de sesión**: Como un usuario de cualquier tipo, quiero poder ingresar mi email y contraseña para poder acceder a la aplicación.

2. **Cierre de sesión** : Como un usuario de cualquier tipo, quiero poder salir de mi cuenta para proteger mi cuenta cuando no la uso.

3. **Invitación de usuarios**: Como un usuario administrador, quiero poder ingresar los emails y roles de otras personas dentro de mi organización para que estas puedan acceder a la aplicación.

4. **Eliminación de usuarios**: Como un usuario administrador, quiero poder eliminar otros usuarios de la aplicación para poder controlar quién tiene acceso a la misma.

5. **Modificación de usuarios**: Como un usuario administrador, quiero poder modificar los emails y roles de otros usuarios de la aplicación para mantener la información actualizada y controlar el nivel de acceso que tiene cada usuario.

6. **Creación de conexiones de datos**: Como un usuario administrador, técnico o experto en calidad quiero poder crear conexiones a fuentes de datos, ya sean relacionales (PostgreSQL o MySQL), no relacionales (MongoDB) o archivos (CSV o JSON), para poder extraer el esquema de las mismas y medir la calidad de los datos dentro de la aplicación.

7. **Modificación de conexiones de datos**: Como un usuario administrador, técnico o experto en calidad quiero poder cambiar el nombre, la descripción o los archivos de las conexiones de datos para poder mantenerlas actualizadas a lo largo del tiempo.

8. **Eliminación de conexiones de datos**: Como un usuario administrador, técnico o experto en calidad quiero poder eliminar una conexión para poder quitar del sistema las que ya no son necesarias.

9. **Refrescar esquema de datos**: Como un usuario administrador, técnico o experto en calidad quiero poder refrescar el esquema extraído de una conexión

de datos para que en los casos donde hayan habido cambios en la conexión, los mismos se vean reflejados en el sistema.

10. **Creación de modelos generales**: Como un usuario administrador, técnico o experto en calidad quiero poder definir modelos generales para poder definir los distintos grupos de dimensiones, factores y métricas que se usan dentro del sistema.

11. **Modificación de modelos generales**: Como un usuario administrador, técnico o experto en calidad quiero poder modificar los contenidos de mi modelo general para mantener la información actualizada.

12. **Eliminación de modelos generales**: Como un usuario administrador, técnico o experto en calidad quiero poder eliminar un modelo general para poder quitar del sistema los que ya no son necesarios.

13. **Creación de métodos**: Como un usuario técnico quiero poder definir un método ejecutable para poder medir mis métricas de calidad.

14. **Modificación de métodos**: Como un usuario técnico quiero poder actualizar un método para poder mantener actualizado el código y parámetros del mismo.

15. **Eliminación de métodos**: Como un usuario técnico quiero poder eliminar un método para poder quitar del sistema código no necesario.

16. **Creación de modelos instanciados**: Como un usuario administrador, técnico o experto en calidad quiero poder definir modelos instanciados para poder asignarle métodos a las distintas métricas de un modelo general y definir en qué conexiones se van a utilizar.

17. **Modificación de modelos instanciados**: Como un usuario administrador, técnico o experto en calidad quiero poder modificar un modelo instanciado para poder mantenerlo actualizado a lo largo del proceso de calidad de datos.

18. **Eliminación de modelos instanciados**: Como un usuario administrador, técnico o experto en calidad quiero poder eliminar un modelo instanciado para quitar del sistema los modelos obsoletos.

19. **Creación de *jobs***: Como un usuario administrador, técnico o experto en calidad quiero poder definir un job para poder programar la ejecución de un modelo instanciado.

20. **Modificación de *jobs***: Como un usuario administrador, técnico o experto en calidad quiero poder cambiar los parámetros de un job para ajustar los detalles de su ejecución según haga falta.

21. **Eliminación de *jobs***: Como un usuario administrador, técnico o experto en calidad quiero poder eliminar un *job* para quitar del sistema las ejecuciones y resultados que ya no son necesarios.

22. **Creación de perfiles de evaluación**: Como un usuario administrador, técnico o experto en calidad quiero poder definir un perfil de evaluación y sus reglas para poder evaluar los resultados de una ejecución de un *job*.

23. **Modificación de perfiles de evaluación**: Como un usuario administrador, técnico o experto en calidad quiero poder cambiar los detalles de de un perfil de evaluación o sus reglas para poder ajustar los parámetros de evaluación del mismo a lo largo del tiempo.

24. **Eliminación de perfiles de evaluación**: Como un usuario administrador, técnico o experto en calidad quiero poder eliminar un perfil de evaluación para quitarlo del sistema junto con todas sus reglas cuando ya no es más necesario.

25. **Notificación de finalización de *job***: Como un usuario de cualquier tipo quiero poder recibir notificaciones al finalizar la ejecución de un *job* para poder saber cuando están disponibles nuevos resultados.

26. **Visualizar resultados de *job***: Como un usuario de cualquier tipo quiero poder acceder a los resultados de las distintas ejecuciones de los jobs para medir y comparar la calidad de mis datos

27. **Aplicar perfiles a resultados de un *job***: Como un usuario de cualquier tipo quiero poder aplicar un perfiles a los resultados de una ejecución de un *job* para poder ver cómo se comparan los resultados con lo esperado por cada regla de evaluación.