UNIVERSIDAD DE LA RIOJA

TRABAJO FIN DE ESTUDIOS

Título

Conceptualización e implementación de un framework de Data Observability para plataformas analíticas de datos modernas

Autor/es

Raúl Benito Martínez

Director/es

CÉSAR DOMÍNGUEZ PÉREZ

Facultad

Facultad de Ciencia y Tecnología

Titulación

Grado en Ingeniería Informática

Departamento

MATEMÁTICAS Y COMPUTACIÓN

Curso académico

2021-22

publicaciones.unirioja.es
E-mail: publicaciones@unirioja.es

UNIVERSIDAD DE LA RIOJA

Facultad de Ciencia y Tecnología

TRABAJO FIN DE GRADO

Grado en Ingeniería Informática

Conceptualización e implementación de un framework de Data Observability para plataformas analíticas de datos modernas

Realizado por:

Raúl Benito Martínez

Tutelado por:

César Domínguez Pérez

Carlos Acedo Nieto

Logroño, enero, 2022

ÍNDICE

RESUMEN

Hoy en día, en proyectos de cualquier ámbito informático, surgen problemas relacionados con la calidad de los datos que manejan continuamente (datos obsoletos, pérdidas de información, cambios en el formato…). Muchas de estas cuestiones pueden ser especialmente problemáticas si no se detectan a tiempo, y a menudo, programas especializados en manejar grandes cantidades de información, no aportan las herramientas apropiadas para poder detectar y gestionar estas dificultades.

No obstante, este tipo de situaciones se pueden prevenir si somos capaces de identificar cuándo hay un problema con la información que estamos utilizando, algo que podemos hacer aplicando Data Observability.

Data Observability es un concepto propio de sistemas software, que se basa en analizar el funcionamiento interno de una aplicación basándonos en algunos parámetros que podemos analizar y transformar en información útil.

El objetivo de este proyecto es implementar una serie de herramientas de Data Observability sobre DBT Cloud (plataforma de transformaciones ELT), que sean capaces de extraer información de los datos que estamos utilizando, además de aportar las herramientas necesarias para poder explotar dicha información, permitiendo a los usuarios corregir este tipo de problemas antes de que comiencen a generar consecuencias.

ABSTRACT

Currently, in projects of any informatics field, problems arise related to the quality of the data that they continuously handle (obsolete data, loss of information, changes in format...). Many of these problems can be especially problematic if they are not detected in time, and programs specialized in handling large amounts of information often do not provide the appropriate tools to be able to detect and manage these problems.

These types of problems can seriously affect the probability of a successful completion of a project, as they can appear spontaneously, deviate planning and generating recurrent problems whose cause is not easy to identify.

However, this type of situation can be prevented if we are able to identify when there is a problem with the information we are using, something that we can do by applying Data Observability.

Data Observability is a concept of software systems, which is based on analyzing the internal functioning of an application based on some parameters that we can analyze and transform into useful information.

The objective of this project is to implement a set of Data Observability tools on DBTCloud (ELT transformation platform), which are capable of extracting information from the data that we are using, in addition to providing the necessary tools to be able to exploit said information, allowing users to correct these issues before they start to generate consequences.

1. INTRODUCCIÓN

En este apartado daremos contexto a los problemas que intentaremos solucionar con este proyecto y analizaremos las soluciones que existen actualmente para ellos. Expondremos también las distintas alternativas que podemos llevar a cabo para crear nuestra propia solución junto a sus pros y sus contras, así como la alternativa por la que nos hemos decantado.

También analizaremos en este apartado la planificación junto con el alcance y los objetivos que debe cumplimentar el producto final.

1.1. Contexto

El origen de este TFG viene de una propuesta por parte de la empresa SDG Group, la empresa donde realicé mi estancia en el periodo de prácticas. La empresa opera en ámbitos relacionados de forma muy cercana con la gestión de grandes cantidades de información. Algunos de estos ámbitos son Business Analytics, Big Data, Data Governance y otros de similar índole.

Las operaciones que suele desempeñar la empresa tienden a involucrar tomar estos datos y transformarlos continuamente: moviéndolos de una tabla a otra, filtrándolos y desechando parte de los registros, reestructurando las tablas que los contienen, etc. Es debido a esto que acaban por aparecer dificultades relacionadas con la calidad del dato. Algunas de estas dificultades son por ejemplo:

- Datos obsoletos que necesitan ser actualizados periódicamente para ser usados.
- Información que se ha perdido durante alguna de las transformaciones que ha sufrido.
- Estructuras de tablas alteradas haciendo que se vean afectadas aplicaciones externas que dependen de la anterior estructura.

Muchas de estas situaciones son gestionables y corregibles, por ejemplo mediante Git (sistema de control de versiones más utilizado a día de hoy), pero la mayor amenaza que generan estos problemas no es el corregirlos, sino tratar de detectar cuándo exactamente los datos se han vuelto problemáticos para así revertirlos. Adicionalmente, también pueden surgir complicaciones a lo largo del tiempo si no hay un mecanismo eficaz para identificar cuándo nuestros datos están fallando, ya que no es una tarea sencilla en todos los casos.

En este tipo de ámbitos las técnicas de Data Observability pueden ser de gran utilidad. A grandes rasgos, estas técnicas permiten identificar en tiempo real la calidad de nuestros datos y si presentan algún tipo de problema, para así tener un mayor conocimiento de la información que estamos utilizando en todo momento, y paliar este tipo de problemas ya mencionados.

Si bien en el mercado ya existen soluciones funcionales y de buena calidad para aplicar Data Observability sobre nuestros conjuntos de datos, la mayoría de estas son de pago, y suponen unos costes bastante elevados, los cuales no son sencillos de justificar en la mayoría de proyectos. También existen soluciones Open Source, pero mi estancia de prácticas estuvo principalmente enfocada a analizar algunas de estas herramientas y a comprobar cómo de útiles pueden llegar a ser, y las conclusiones fueron que: Si bien las herramientas Open Source dedicadas a Data Observability pueden parecer muy prometedoras y pueden tener mucho potencial a largo plazo, a corto plazo generan

muchos problemas al no ser productos finales la mayoría de ellos, por lo que pueden acabar generando más dificultades de las que realmente resuelvan.

Es aquí donde entra este TFG, el cual puede aportar una solución capaz de solventar los problemas mencionados previamente, sin necesidad de tener más que la funcionalidad necesaria para lidiar con ellos, estando además enfocado a las herramientas que SDG más utiliza y donde más veces se necesita aplicar Data Observability.

1.2. Data Observability

Antes de que desarrollemos el trabajo, es fundamental que hagamos una breve introducción a Data Observability, para poder así comprender mejor qué tipo de solución necesita cubrir nuestro TFG.

Se entiende por Data Observability la capacidad que tiene una organización para comprender el estado de los datos de su sistema, así como la calidad de los mismos. Las técnicas que se usan para aplicar esta técnica tienden a estar relacionadas con la monitorización (automática o manual) de los sistemas de información junto con los datos que contienen.

Habitualmente se suele decir que Data Observability se sostiene sobre cinco pilares, los cuales sirven para identificar los distintos controles que se deben llevar sobre los sistemas. Los pilares son los siguientes:

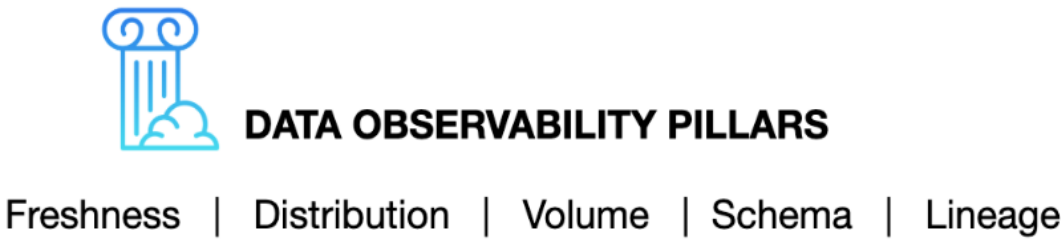

Ilustración 1 Pilares de Data Obs, por Barr Moses

- **Freshness**: Este pilar hace referencia a cómo de recientes son los datos. La información desactualizada tiende a ser una de las principales causas de fallos en el flujo de datos, por lo que es muy importante controlar las fechas de actualización.
- **Distribution**: Se centra en el análisis de los datos mediante la aplicación de métricas, como por ejemplo, el porcentaje de registros nulos, o la media. Un conocimiento apropiado de qué rangos deberían abarcar ciertas métricas, puede permitir identificar fallos en campos que de cualquier otra forma resultarían opacos para el usuario.
- **Volume**: El volumen maneja la cantidad de registros que estamos utilizando. Si conocemos el total de registros de una tabla y ese total varía cuando no debería haberlo hecho, será fácil realizar una comparativa y encontrar los registros faltantes o sobrantes.
- **Schema**: Este pilar apunta hacia la estructura que deben tener los datos, es decir, que los datos numéricos sean números, que las fechas tengan el formato apropiado o que las cadenas de texto no tengan más caracteres de los especificados en su definición.
- **Lineage**: El pilar del linaje es el más alejado de los propios datos, ya que es el que hace referencia a los metadatos generados del procesado de los propios datos.

Un ejemplo sería la fecha de ejecución de un proceso, los archivos a los que se acceden o los ficheros generados tras la ejecución.

Esta definición de cinco pilares viene dada por Montecarlo, una de las soluciones de Data Observability más prominentes del mercado, la cual cubriremos en la próxima sección.

Nuestro proyecto se centrará principalmente en los pilares de **Freshness**, **Distribution** y **Volume**, aunque es destacable el hecho de que se ha desarrollado un proyecto paralelo a este centrado en el análisis del **Lineage**, pero nuestra intervención en ese proyecto es anecdótica, por lo que no entraremos en mayor detalle.

1.3. Antecedentes

En esta sección haremos un recorrido por las distintas soluciones que hay actualmente en el mercado, para poder hacernos una mejor idea de qué métodos emplean para atajar los problemas analizados previamente.

Los tres productos en los que más nos hemos fijado a la hora de establecer el alcance del proyecto y de realizar el desarrollo han sido Re_data, Datafold y Montecarlo. Los presentaremos en función de cómo de completos y cuánta funcionalidad tienen las plataformas, empezando por Re_data, la cual es la menos madura y terminando por Montecarlo, el cual es posiblemente el producto de Data Observability más completo y maduro del mercado actual.

1.3.1. Re_data

La primera de las herramientas que vamos a mencionar será Re_data, un framework de confianza de datos que se aplica sobre la herramienta de transformación de datos DBT. Re_data tiene soporte varias de las bases de datos más usadas en el mercado, siendo algunas de estas Snowflake, Postgre, Amazon Redshift o Databricks (daremos una explicación más extensa de las herramientas relevantes para el proyecto en las siguientes secciones).

El hecho de que Re_data solo funcione sobre DBT, puede ser un factor limitante, pero en el caso de la empresa SDG, DBT es la principal plataforma de transformación de datos y de gestión de SQL que utilizan, por lo que esto no supone ninguna complicación, además, el hecho de que funcione con la mayoría de bases de datos y warehouses, hace que sea bastante extensible una vez hemos decidido comprometernos con DBT.

Otra particularidad de Re_data es que es Open Source, lo cual viene con una serie de ventajas bastante relevantes, principalmente el hecho de que es gratuita, y la posibilidad de acceder al código para poder ver cómo han implementado la funcionalidad de la herramienta.

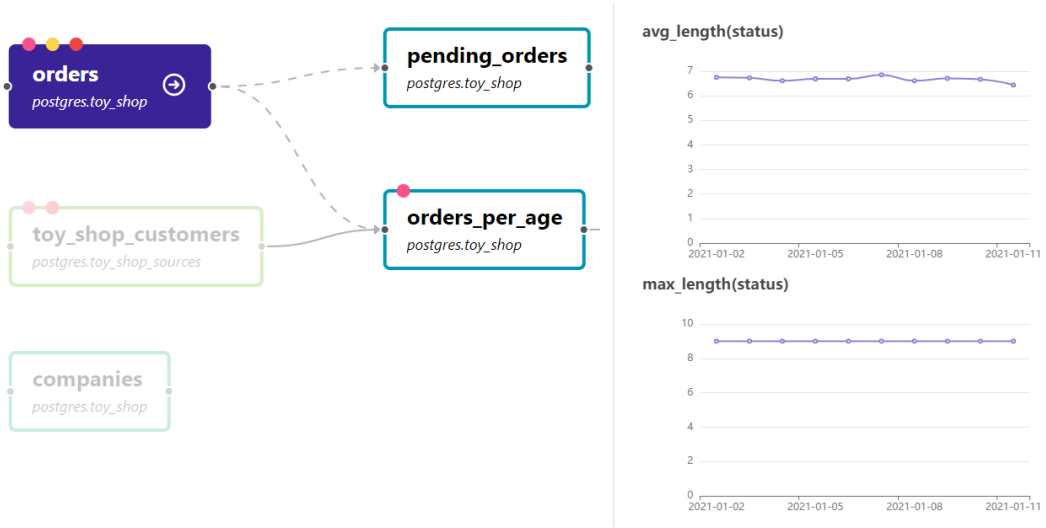

Ilustración 2 Interfaz Re_data

Otro punto positivo es que la interfaz de Re_data es muy limpia visualmente (ver Ilustración 2), mostrando toda la información necesaria en pantalla de forma clara y concisa, además de actualizarla de forma dinámica cuando los datos observados varían.

Por lo que hemos podido apreciar, los pilares que Re_data analiza son los de Freshness, Distribution y Lineage, pero está centrada principalmente en la extracción de análisis y métricas, es decir, en Distribution. Por desgracia, las métricas que se analizan son poco numerosas y puede ser que se queden algo cortas, ya que solo se analizan las más básicas como el máximo, el mínimo, la media y similares.

1.3.2. Datafold

La siguiente solución es Datafold, un sistema de control de observabilidad orientado a encontrar problemas en los datos a más bajo nivel, dando la posibilidad de no solo analizar las métricas de los datos a nivel de tabla, sino también a nivel de columna, lo que implica que es capaz de identificar registros problemáticos individuales dentro de una celda de una tabla.

Datafold no es una solución Open Source a diferencia de Re_Data, y está orientada a que lo utilicen empresas con una capacidad adquisitiva mayor; es por ello que, presenta mayor funcionalidad y profundidad que la anterior herramienta.

Funciona sobre las mismas bases de datos que Re_Data, pero además de esto, la información resultante de este proceso de observabilidad no solo se puede procesar y orquestar mediante DBT, sino que también puede aprovecharse mediante Airflow, una de las soluciones de orquestación de procesos más prominentes en el mercado actual.

Adicionalmente, también permite registrar las alertas pertinentes mediante aplicaciones externas y no solo mediante la interfaz visual de Datafold, dando la posibilidad de registrar estas incidencias en Slack, PagerDuty o vía E-mail.

Con respecto a su interfaz, es más completa que la de Re_Data y presenta más funcionalidad, aunque es menos visual y esto hace que sea algo más complicado identificar la información que necesitamos rápidamente (ver un ejemplo en Ilustración 3).

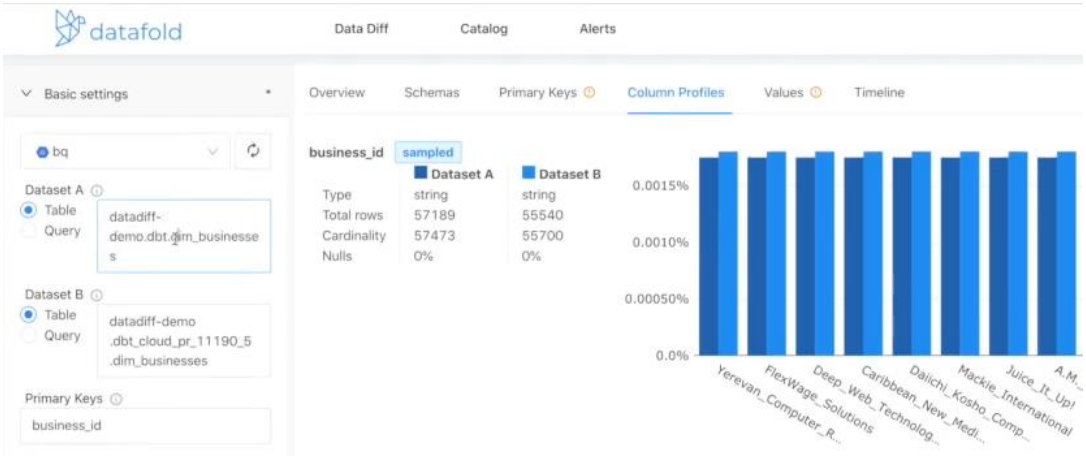

Ilustración 3 Interfaz Datafold

1.3.3. Montecarlo

Para terminar, hablaremos ahora de Montecarlo. Esta solución es una de las primeras que se propusieron solventar el problema de la observabilidad de datos, y pudo sentar unas bases (los cinco pilares) en las que otras herramientas posteriores se apoyarían a la hora de identificar la información que debe analizarse y cómo debe hacerse.

El enfoque de cinco pilares de Montecarlo permite diferenciar claramente los tipos de test que debemos llevar a cabo sobre nuestros datos, y en caso de que alguno falle, se facilita mucho el localizar los datos problemáticos y que tipo de fallos presentan.

Esta solución puede integrarse con todas las mencionadas previamente, y otras tantas más bases de datos y herramientas de manejo de datos como Spark, Looker o Hive. No solo esto, sino que algunas de ellas presentan una interfaz de Montecarlo dentro de las propias aplicaciones, sin necesidad de manejar un cliente externo.

La interfaz de esta herramienta está bastante fragmentada (ver Ilustración 4), dando una vista de pájaro inicial para identificar las distintas tablas y bases de datos que estamos manejando, y permitiéndonos descender a bajo nivel progresivamente hasta localizar el nivel de granularidad que necesitamos (de base de datos a tabla, de tabla a columna, de columna a campo).

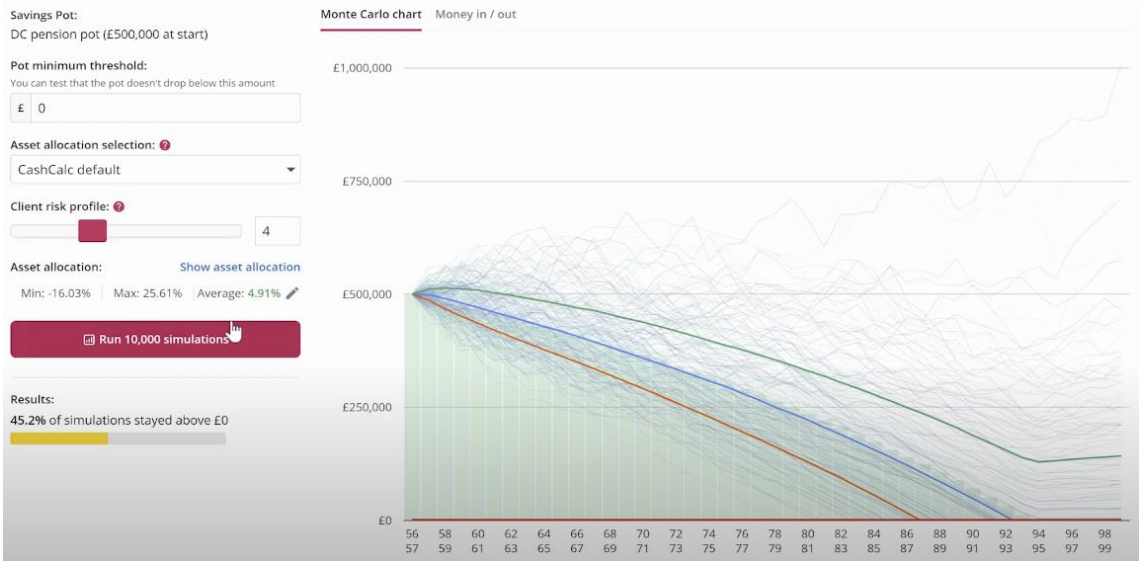

Ilustración 4 Interfaz Montecarlo

Al igual que Datafold, Montecarlo es una solución de pago que se adapta a las necesidades de empresas medianas y grandes principalmente, por lo que el código que emplean para realizar estos procesos no está publicado, lo que implica que solo podremos tomar ideas de esta plataforma a nivel estructural.

1.4. Enfoque

Una vez estudiadas las aplicaciones ya existentes, podemos identificar los puntos fuertes y débiles de cada una de las soluciones ya existentes en el mercado, por lo que podemos empezar a hablar de qué tipo de solución vamos a dar nosotros.

Está claro que el enfoque de cinco pilares de Montecarlo es una estratificación idónea de la información, por lo que analizar estos pilares e identificar cuáles de ellos nos pueden interesar es una labor prioritaria.

Algunos de estos pilares pueden ser resumidos en un solo campo, como el volumen como cantidad de registros o el freshness para la antigüedad de los datos, por lo que es relativamente sencillo recoger estos datos y dan bastante contexto a la información que vamos a representar.

También está el schema, el cual puede ser más complicado o más sencillo dependiendo de si la base de datos que estemos usando da soporte para extraer el schema de una tabla vía query. En el caso de Snowflake, tenemos esta funcionalidad disponible, por lo que también nos interesa aplicar este pilar.

Con distribution la situación se vuelve más compleja, ya que no puede resumirse en una sola métrica, sino en un conjunto que analicen distintos aspectos diferentes de los datos. Las métricas que se analizan en distribution por lo general se aplican sólo sobre campos numéricos, así que nos centraremos en estos y aplicaremos sólo las métricas que se puedan sobre campos de tipo cadena de caracteres.

Para terminar con los pilares tenemos el lineage, el cual se enfoca en la extracción de datos a nivel de proyecto. En el caso de este proyecto, este pilar fue el enfoque principal del proyecto paralelo mencionado con anterioridad, por lo que nosotros nos centraremos en los otros cuatro pilares y dejaremos este apartado.

Una vez hemos definido los pilares que vamos a aplicar, podemos deducir de ellos el tipo de métricas que vamos a analizar, así como la información que vamos a aportar a los usuarios.

Visto esto, necesitaremos que nuestra solución sea capaz de extraer las métricas pertinentes de los datos, plasmar esta información y añadir test que puedan indicar cuando algunos registros no son correctos.

Para la interfaz visual, trataremos de acercarnos a la que presenta Montecarlo, buscando presentar los datos primero a nivel alto, y permitiendo al usuario ir descendiendo de nivel para que pueda apreciar la información en el nivel de granularidad que necesite.

Para las plataformas para la integración, dado el tiempo limitado que tenemos nos centraremos en las necesarias en lugar de tratar de abarcar todas las grandes plataformas del mercado, dejando esta tarea para futuras iteraciones del proyecto.

1.5. Herramientas

Una vez vistos algunos ejemplos de herramientas de data observability, necesitaremos pensar cómo vamos a conseguir unos resultados similares con los recursos a nuestro alcance, por lo que vamos a empezar por definir a qué plataformas vamos a aferrarnos para llevar a cabo el ejercicio.

Para empezar necesitaremos una base de datos para guardar tanto la información que usemos de ejemplo como la información que nuestra aplicación extraiga. La plataforma seleccionada será **Snowflake**, una base de datos orientada a la nube pensada para almacenar y mover grandes cantidades de información. Esta plataforma proporciona una versión de prueba de un mes sin apenas limitaciones y además viene con bases de datos a modo de ejemplo, por lo que podemos utilizarla sin mayores complicaciones.

También necesitaremos una herramienta que nos permita extraer la información de Snowflake, aplicar las métricas, y localizar los registros erróneos. Usaremos **DBT** para esta labor: una solución de transformaciones de datos orientada a SQL para bases de datos como Snowflake. La idea es generar el código SQL en DBT y que este se encargue de enviarlo a Snowflake para que se ejecute en dicha plataforma. Esta tarea podría llegar a efectuarse en el propio Snowflake, ya que proporciona soporte para consultas bastante complejas, pero DBT permite combinar el uso de SQL con Jinja, un lenguaje de templating que permite inyectar código dinámico sobre código SQL, reduciendo en gran medida el tamaño de las consultas y añadiendo la posibilidad de adaptar el código de cada una de estas a cada tabla con interacciones mínimas por parte del usuario.

Para terminar, queremos representar de la forma más visual y sencilla posible la información extraída de Snowflake mediante DBT, y a su vez queremos tratar de dar un enfoque similar al de Montecarlo a la hora de representar esta información; es por ello que usaremos **PowerBI** para esta tarea, una herramienta de dashboarding pensada para extraer registros de bases de datos de forma automática y plasmarlos en gráficos y tablas interactivos. Más adelante veremos que este programa tiene algunas limitaciones, pero para una aproximación inicial es suficiente, ya que si quisiéramos algo más funcional, necesitaríamos programar nuestra propia interfaz visual, lo cual es inviable dado el tiempo disponible para el proyecto.

1.6. Planificación

1.6.1. Alcance

Definidos ya el enfoque y las herramientas que vamos a usar, vamos ahora a indicar los requisitos que la aplicación debe cumplimentar.

Código	Descripción
RE-01	Se deben poder manejar grandes cantidades de información de forma dinámica.
RE-02	El usuario debe poder configurar la aplicación según sus necesidades.
RE-03	Se deben poder tratar registros de todos los tipos de datos aceptados en Snowflake
RE-04	La aplicación debe admitir cambios en la configuración entre ejecuciones.
RE-05	Los registros de la base de datos deben representarse automáticamente en PowerBI sin necesidad de interacción por parte del usuario.
RE-06	El proceso debe estar lo bastante automatizado para que el usuario no necesite más conocimientos que los requeridos para configurar la ejecución.
RE-07	La configuración de las ejecuciones debe estar contenida en un solo archivo dentro de DBT.
RE-08	Debe haber métricas asociadas a los pilares de Freshness, Schema, Volume y Distribution.

Vamos ahora a desarrollar en mayor profundidad los requisitos recogidos:

- **RE-01:** Las aplicaciones de Data Observability están orientadas a ayudar a programas que manejan grandes cantidades de datos, por lo que las querys resultantes de nuestros procesos deben estar optimizadas para ser lo más rápidas posibles.
- **RE-02:** Debe haber algún mecanismo de configuración que permita al usuario especificar qué tablas quiere analizar, qué métricas necesita, a qué columnas quiere aplicar dichas métricas etc.
- **RE-03:** Aunque el programa esté centrado en métricas numéricas, tiene que poder aplicar a campos no numéricos como Varchar o Date todas las métricas posibles como el Null-rate.
- **RE-04:** En caso de que el usuario realice cambios en la configuración de una ejecución a otra, no se deben perder los registros nuevos que deben ser compatibles a nivel de columnas con los anteriores.
- **RE-05:** El usuario no debería necesitar configurar PowerBI más allá de añadir alguna vista adicional a las dadas para que la información de Snowflake se refleje.
- **RE-06:** Una vez configurado el proyecto, el usuario no debería tener que hacer nada más que ejecutar los comandos pertinentes para generar los modelos que

se transformarán en bases de datos e introducir los comandos predefinidos para ejecutar el proyecto.

- **RE-07:** Para facilitar el proceso de configuración, toda la configuración de observabilidad del proyecto debe especificarse en un solo archivo.
- **RE-08:** Debe haber métricas de antigüedad de los datos, cantidad de registros, esquema de las tablas y todas las métricas posibles asociadas a Distribution (Null-rate, máximo, mínimo, Z-score, etc).

1.6.2. Cronograma

En esta sección marcaremos en una tabla las distintas porciones del trabajo en función en qué semana vamos a estar avanzando cada una de ellas. La semana cinco se corresponde con la semana del 24 al 30 de enero, mientras que la veinte se corresponde con la del 9 al 15 de mayo.

Hitos	S5	S6	S7	S8	S9	S10	S11	S12	S13	S14	S15	S16	S17	S18	S19	S20
T-1 Análisis																
T-2 Diseño																
T-3 Implementación																
T-4 Interfaz gráfica																
T-5 Memoria																

- **T-1:** La fase de análisis del proyecto abarca todo lo relacionado con buscar información sobre el problema que deseamos resolver e investigar qué soluciones existentes hay en el mercado, así como localizar cuáles son las herramientas que nosotros podemos utilizar y que mejor nos van a servir para crear nuestra propia solución.
- **T-2**: Se inicia una vez se ha establecido qué problema vamos a solucionar y qué herramientas usaremos para ello. Una vez las tenemos, se debe realizar una organización de dichas herramientas y plantear qué funciones usaremos en cada una de ellas para crear nuestro propio framework.
- **T-3**: Una vez creado el esquema de lo que necesitamos en la parte de diseño, será en la fase de implementación donde crearemos el framework que hemos analizado. En esta fase pueden llegar a identificarse algunas limitaciones o mejores aproximaciones que no se habían detectado en las anteriores dos fases, por lo que se contempla que se pueden llegar a producir cambios sobre el diseño planteado inicialmente.
- **T-4**: Ya terminada la parte interna del framework, se pasa a crear una interfaz visual para facilitar el uso de la aplicación. Es fundamental dejar esta parte para el final ya que si queremos que el diseño de la aplicación sea flexible en función de nuestras necesidades en la fase de implementación, nos interesa que la interfaz no esté vinculada a la fase de diseño sino que lo esté al resultado de la fase de implementación.
- **T-5**: Este punto se corresponde con la redacción de la memoria del TFG. Es importante destacar que aunque la memoria se inicia en la semana 18, se deben haber recogido notas de la realización del proyecto para evitar la pérdida de información relevante, y se debe tener preparada la planificación aunque no esté recogida por escrito.

1.6.3. Estimación de tiempos

Es importante establecer también los tiempos estimados que vamos a invertir en cada parte del proyecto. Nos basaremos en la división de fases realizada en el apartado anterior para ello. Adicionalmente, realizaremos una comparación de las horas estimadas con respecto a las horas reales invertidas en cada fase al final del proyecto.

Como se puede apreciar en la siguiente tabla, las partes más costosas en términos de tiempo serán las de implementación y análisis. El motivo de la implicación a la fase de análisis es debido al hecho de que vamos a tratar de dar una solución basada en Data Observability, ámbito en el cual no estamos especializados, por lo que será necesario una investigación exhaustiva para entender qué problemas debemos resolver y cómo deberíamos hacerlo.

Códigos	Fase	Horas
T-1	Análisis	60
T-2	Diseño	40
T-3	Implementación	120
T-4	Interfaz gráfica	30
T-5	Memoria	50
TFG	**Total proyecto**	**300**

Por otro lado, es más sencillo identificar por qué la parte de implementación nos va a llevar tantas horas, ya vamos a estar trabajando con herramientas con las cuales no tenemos experiencia previa, lo cual puede llevar a inversiones de tiempo mayores para tareas que con conocimientos previos podrían llevar a ser simples, y también pueden surgir problemas relacionados con las limitaciones de la aplicación que nos obliguen a modificar el diseño que habíamos establecido previamente.

1.6.4. Gestión de riesgos

Es fundamental para garantizar el éxito del proyecto identificar las posibles amenazas que puedan alterar el cronograma y hacer que haya problemas relacionados con la gestión del tiempo. Para ello, trataremos de analizar algunos de los riesgos más probables que pueden llegar a afectar a la correcta realización del TFG.

1. **RI-1 Curva de aprendizaje demasiado lenta:** Debemos tener en cuenta que vamos a estar en un ámbito con el que apenas tenemos experiencia, por lo que necesitamos hacer un buen análisis inicial de las herramientas para usar las mínimas posibles sin comprometer las características finales del proyecto.
2. **RI-2 Enfermedad:** Somos el único agente que trabaja en el proyecto, por lo que en caso de caer enfermo, el avance se detendrá por completo temporalmente. Para mitigar este riesgo, debemos asegurarnos de mantener una buena gestión del tiempo con respecto a las horas estimadas para poder tener margen de tiempo en caso de necesitarlo.
3. **RI-3 Decisiones erróneas de diseño:** Como bien hemos mencionado antes, al estar manejando programas que no conocemos previamente, puede ser que nos encontremos intentando realizar alguna función para la cual el programa no esté preparado, degenerando en tener que buscar una alternativa o alterar el diseño inicial. Para evitarlo, debemos ser lo más rigurosos posibles en la fase de diseño, tratando de

identificar previamente si las funciones que necesitaremos serán posibles con los programas que tenemos a nuestra disposición.

4. **RI-4 Pérdidas de datos:** Consideramos en este riesgo pérdidas tanto de código fuente como de la memoria. Puede darse al corromperse la memoria donde tengamos guardados estos datos o por fallo humano. Para evitarlo, tendremos copias de todo el código y de la memoria tanto en local como en la nube, las cuales actualizaremos periódicamente.
5. **RI-5 Imposibilidad de realizar requisitos:** Este tipo de problemas son una extensión de RI-3, incluyendo además problemas obviados a la hora de realizar la fase de análisis. Puede ser que uno de los requisitos que hemos establecido sea imposible de efectuar con los programas que manejaremos, en este caso, la solución será recortar ese requisito en caso de que no sea posible efectuarlo sin tener que dejar de lado uno de los programas que estamos utilizando.
6. **RI-6 Mala gestión del tiempo:** Este riesgo puede darse por haber estimado tiempo de menos a una de las tareas, comprometiendo la realización en la fecha correcta de las siguientes. Debido al límite estricto de horas que tiene el trabajo, estaremos atentos al calendario y a las horas invertidas a cada sección para evitar salirnos del marco preestablecido, y en caso de que invirtamos más horas en alguna sección, trataremos de pensar cómo podemos ahorrar esas horas en secciones posteriores.
7. **RI-7 Problemas de entendimiento con el cliente:** El cliente tiene la última palabra en caso de que haya dudas de por dónde seguir avanzando o qué decisiones tomar, por lo que para evitar tener que rehacer porciones del proyecto para que se adecúen a lo que el cliente desea, mantendremos una comunicación activa y casi diaria con SDGGroup, reduciendo al mínimo así el tener que rehacer secciones.

1.6.5. Metodología

Antes de comenzar el proyecto es fundamental saber qué tipo de metodología de trabajo emplearemos. En este caso hemos considerado que la mejor manera de avanzar será empleando desarrollo un en **cascada**, avanzando una a una por las fases establecidas en el apartado 1.6.3 y tratando de solo pasar a la siguiente cuando cada una de las fases anteriores esté terminada. En caso de que encontremos problemas en la etapa actual o en las anteriores, daremos un paso atrás para corregirlos cuando los identifiquemos.

Como particularidad, hemos establecido con SDGGroup que para asegurarnos de que el proyecto avanza en la dirección adecuada y cumple los requisitos establecidos, se realizarán reuniones cortas diarias (aproximadamente tres por semana) en las que se comentarán los avances de días previos y las bifurcaciones que resulten tener en que tomar decisiones de diseño.

2. ANÁLISIS

Como bien hemos mencionado previamente, la fase de análisis será una de las más críticas en la realización del TFG; esto es debido a que vamos a estar usando tecnologías con las que tenemos poca o ninguna experiencia, por lo que es fundamental realizar un análisis profundo de todos los ámbitos que van a intervenir en el proyecto para evitar desviaciones severas a posteriori.

Por ello, las primeras semanas del proyecto estuvieron dedicadas a obtener una mejor comprensión de las técnicas de Data Observability, así como de analizar herramientas ya existentes en el mercado que den soluciones a los problemas más típicos del ámbito; todo esto mientras paralelamente se estudiaba qué clase de solución podemos proporcionar nosotros dada la información que estábamos obteniendo y los recursos a nuestra disposición.

Fue en esas semanas también cuando adoptamos la estructura de cinco pilares de Data Observability para realizar una división de nuestros esfuerzos en cada ámbito más efectiva.

2.1. Planteamiento inicial

La primera semana del TFG estuvo totalmente orientada a buscar qué tipos de problemas había con relación a Data Observability, y qué tipo de solución podíamos producir dado nuestros conocimientos técnicos y tiempo disponible. La primera corriente de pensamiento fue algo distinta a la empleada en el planteamiento final, ya que estaba más orientada a usar una aplicación orientada íntegramente a Data Observability y extenderla.

Esta aplicación es la ya mencionada previamente, **Re_Data**. El hecho de que esta herramienta tuviese una planteamiento muy interesante pero que todavía estuviese en etapas tempranas de desarrollo, hacía que pudiésemos tomarla como una base sólida para el inicio del proyecto, y a su vez que hubiese bastantes direcciones distintas por las que mejorar la aplicación. Esto sumado al hecho de que es Open source, daba un escenario ideal como punto de partida.

No obstante, tras algo de investigación y profundización con la herramienta, el cliente no estaba del todo satisfecho con las capacidades que ofrecía, ya que las primeras interacciones con Re_Data no fueron tan sencillas ni tan enriquecedoras como parecía que iban a serlo en primera instancia, principalmente por complicaciones relacionadas con discrepancias entre la documentación del programa y el uso real del mismo.

Estas discrepancias resultaron ser bastante severas y nos instaban a pensar que podían seguir ocurriendo a menudo a lo largo del proyecto, lo cual ralentizaba mucho la ejecución del mismo solo tratando de hacer funcionar el programa de la forma que lo necesitamos; es por ello que optamos por distanciarnos de esta aproximación, y decidimos realizar un cambio de enfoque.

2.2. Nuevo enfoque

Vistas las complicaciones que pueden surgir al tratar de extender Re_Data, se propuso un cambio en el planteamiento, el cual implicaba dejar de lado el intentar extender soluciones ya existentes, y generar nuestro propio framework de forma más manual, y empleando programas menos especializados y más versátiles que nos permitan una mayor flexibilidad en nuestra tarea.

La base de esta idea viene de que la empresa SDG ya había realizado previamente un proyecto relacionado con Data Observability, el cual se centraba explícitamente en el pilar del Lineage. Dicho proyecto fue realizado íntegramente en la herramienta de transformación de datos **DBT**, usando como base de datos **Snowflake**. La mera existencia de dicho proyecto era el punto de inflexión que nos hizo querer enfocar nuestro TFG en esa dirección, debido a que al existir un precedente previo a la aplicación de Data Observability usando esas dos herramientas, queda implícito que hay posibilidades muy altas de que podamos producir con éxito un framework usando esas plataformas.

Debido a que la empresa ya tenía ese proyecto como solución orientada al linaje, nuestra solución trataría de estar centrada en los otros cuatro pilares: **Freshness, Schema, Volume y Distribution**. Sabiendo esto, solo nos quedaba identificar el si era posible realizarlo con las mismas herramientas que el proyecto paralelo.

2.3. Capacidades necesarias

Para saber si DBT y Snowflake son adecuados para la tarea a realizar, es necesario primero identificar apropiadamente qué tipo de funciones necesitaremos que estos programas sean capaces de llevar a cabo. Para ello, nos fijamos nuevamente en la estructura de pilares.

Para empezar, debemos destacar que por la forma que tiene de funcionar DBT, las capacidades limitantes para la realización del proyecto estarán casi todas en Snowflake. Ya que DBT a grandes rasgos solo prepara el código de SQL para que Snowflake lo ejecute, las funciones específicas de cada base de datos se dan solo a nivel de Snowflake. Es por ello que deberemos revisar principalmente las funciones que dicha BD proporciona además de las capacidades propias del lenguaje SQL.

El primer pilar que analizaremos será el de **Freshness**, que representa cómo de recientes son los datos. Es sencillo ver qué clase de función necesitaremos para mantener este pilar controlado: Necesitaremos algo que nos permita conocer cuál ha sido la última actualización de datos que se produjo en una tabla. Snowflake nos proporciona una sencilla función llamada **LAST_LOAD_TIME()**, la cual nos devuelve la última fecha de actualización asociada a la tabla pertinente, por lo que sabiendo esto, este campo podremos tenerlo controlado en todo momento sin necesidad de realizar código complejo.

El próximo pilar al que buscaremos dar solución será el **Schema**. Para ello, necesitaremos una manera de saber cuándo ha sido modificado el schema de la aplicación. Para este pilar fue más complicado encontrar una función que nos sirva para gestionarlo, pero acabamos por encontrar algo: Snowflake tiene una función específica para extraer los schemas de una tabla como DLL. Dicha función es **GET_DDL("STRUCTURE_TYPE", "TABLE_NAME")**, la cual dada una estructura de datos (en este caso "table") y el nombre de una tabla, nos devuelve el DDL (Data Definition Language: Estructura de columnas que componen una tabla en SQL) a modo de cadena de caracteres. Esto implica que si guardamos este valor en un campo de una tabla y mantenemos esos registros en un histórico, podremos comprobar si dicho registro ha cambiado a lo largo del tiempo y de qué manera exactamente.

Es muy importante destacar que para dar solución al pilar del Schema, se vuelve necesario el mantener los registros de las métricas en un **histórico**, así como el poder asociarles una **fecha** a cada uno de ellos, por lo que necesitaremos saber si esto es posible con nuestras herramientas actuales.

Empezando por el histórico, solo sería necesario pensar una estructura de datos incremental, de modo que a cada tabla de métricas se añaden registros con cada ejecución de nuestra solución, en lugar de sobrescribir los registros ya existentes. Esto se puede gestionar fácilmente a nivel de DBT, ya que posee unos parámetros de configuración que se asignan a cada modelo (las estructuras las cuales se transforman en tablas de Snowflake). Dichos parámetros nos permiten seleccionar si queremos que cada vez que se ejecute el código SQL del modelo, se sobreescriben los registros de la tabla donde se materializa la información, o solo se añaden los registros nuevos como nuevas filas.

Para asociar una fecha a cada registro según cuando fue generado, lo tenemos bastante sencillo gracias a la función de **current_timestamp()** de Snowflake, la cual devuelve la fecha exacta actual en formato Timestamp, por lo que si tenemos una columna asociada a cada fila donde apliquemos dicha función, sabremos la fecha exacta a la que cada registro se creó, permitiéndonos filtrar o identificar cuándo ha habido cambios en ciertas métricas.

Volviendo a los pilares, sigue ahora el de **Volume**. Este pilar es el más sencillo de todos, ya que solo necesitaremos realizar un **COUNT** a la tabla objetivo para conocer en ese momento la cantidad de registros totales que tiene. En este caso no es necesario buscar funciones específicas que proporcione Snowflake ya que con las que nos brinda SQL es suficiente.

Para terminar con los requisitos impuestos por cada pilar, tenemos el de **Distribution**. Este pilar es el más abierto de todos, ya que abarca cualquier información que nos indique patrones o estadísticas en los datos; es decir, **métricas estadísticas** principalmente. Algunas de estas métricas pueden ser la media, la desviación, mínimo, máximo, Z-Score… Todas estas métricas mencionadas y otras muchas más son posibles de calcular mediante funciones básicas de SQL, o mediante la aplicación de fórmulas estadísticas que también podemos replicar con SQL. No será necesario realizar un análisis en profundidad de qué métricas podremos aplicar y cuáles no, ya que una vez sabemos que es posible gestionar Distribution, el cómo lo haremos lo dejaremos para el apartado de **Diseño.**

2.4. Conclusiones del análisis

Una vez hemos averiguado qué tipo de funciones necesitamos y qué nos puede aportar tanto Snowflake como DBT, llegamos a la conclusión de que en efecto, el proyecto es posible de realizar usando estas dos herramientas, y sabiendo que podemos dar solución a los cuatro pilares de mejor o de peor manera, ya podemos llegar a un punto seguro donde tenemos la garantía de que existe la posibilidad dar una solución final con toda la funcionalidad que necesitamos.

Esto no implica que no vayan a surgir complicaciones que puedan desviar el proyecto, ralentizando y forzándonos quizás a invertir menos tiempo y esfuerzos de lo que nos gustaría en algunos apartados como vimos en el análisis de riesgos, pero el tener esta información junto con el precedente de que ya se ha realizado un proyecto previo en la empresa que guarda algo de relación con el nuestro, nos permite tener suficiente seguridad como para poder avanzar a la siguiente fase.

Destacamos también que no hemos incluido en la fase de análisis **PowerBI**, el programa que usaremos para mostrar de forma visual nuestro proyecto, ya que solo necesitaremos una interfaz simple que muestre de forma algo más visual toda la información que plasmaremos en tablas, y con la experiencia previa que poseo de PowerBI (escasa pero suficiente), sumado al hecho de que el programa puede conectarse directamente a Snowflake y extraer la información de las tablas en tiempo real; es suficiente para saber que este programa visual no será un factor limitante para el proyecto y que podremos usarlo una vez tengamos listas las tablas.

3. DISEÑO

Una vez sabemos qué herramientas vamos a utilizar, en esta sección explicaremos la fase de diseño, en la cual decidiremos con exactitud qué tipo de servicios va a proporcionar nuestra aplicación, además de planificar cómo vamos a hacer para que funcione a nivel interno.

Para realizar esta tarea, trataremos de dividir el diseño en distintas porciones, para así poder identificar con mayor facilidad cuáles van a ser los objetivos que tendremos en cada parte, así como los medios que necesitaremos utilizar para alcanzarlos.

Comenzaremos por la división más evidente: Las tres aplicaciones que vamos a utilizar; y luego pasaremos a otros tipos de divisiones menos relacionadas con la parte física de la aplicación y más relacionadas con la conceptual, como es el estudio de las métricas y las capas que tendrá la aplicación.

3.1. Funciones de cada aplicación

Lo primero antes de planificar cómo vamos a cumplir los objetivos impuestos previamente, será bajar un nivel de granularidad y decidir qué función va a tener que desempeñar específicamente cada una de las aplicaciones que participará en nuestro proyecto, así como definir hasta qué punto vamos a necesitar redactar código para que desempeñen dichas funciones.

3.1.1. Snowflake

Comenzaremos hablando de la primera aplicación del proceso. Snowflake tendrá la función de ser nuestra **base de datos**, lo que implica que todos los registros que usemos como punto de partida, junto con los que generemos con la aplicación, se deberán almacenar aquí.

Lo que tendremos que hacer en Snowflake será ínfimo, ya que posee una serie de bases de datos con registros por defecto los cuales serán suficientes para realizar nuestro proyecto, por lo que la única interacción que tendremos que realizar será la de crear una base de datos con un nombre y unos esquemas apropiados para las operaciones que realizaremos

3.1.2. DBT

Será en esta aplicación donde tendremos que centrar nuestros mayores esfuerzos a la hora de realizar el código que hará funcionar el proyecto, puesto que no solo deberemos crear los procesos que lleven a cabo todas las operaciones de cálculo de métricas, sino que deberemos también configurar cada uno de los modelos que desencadenan estas operaciones para generar los registros pertinentes en cada una de las tablas correspondientes, y así poder almacenar esta información a medio y largo plazo.

Recogeremos en el siguiente listado todas las funciones que tendremos que programar manualmente para que DBT sea capaz de realizar el papel que le hemos asignado:

- Acceder a los registros que usaremos como fuente almacenados en Snowflake.
- Crear un fichero de configuración donde podamos configurar qué tablas queremos analizar y qué métricas queremos extraerles.
- Programar los códigos SQL que extraerán las métricas de las tablas de origen de forma estática.

- Crear una serie de macros que actúen a modo de funciones y que adapten los códigos SQL previos para que se ejecuten de forma dinámica sobre los modelos seleccionados.
- Crear los propios modelos que se encargarán de llamar a las macros previas para que ejecuten los códigos SQL con la configuración correspondiente según el archivo de configuración.
- Estructurar la información resultante para que se guarde en tablas incrementales o que se sobreescriba en función de qué tipo de tabla queremos crear.

Adicionalmente, añadiremos una función extra que aunque no es necesaria para que la aplicación funcione, proporcionará una mejora de calidad de vida bastante alta al usuario final: Ya que el paso de crear los modelos que llamen a las funciones será muy similar independientemente de qué modelo creemos, y la única variación será las distintas parametrizaciones que incluyamos en el archivo de configuración, crearemos un script de **python** el cual tomará el mencionado archivo de configuración, lo leerá, y creará de forma automática todos los modelos necesarios, escribiendo en cada uno de ellos las llamadas a las macros con los parámetros especificados en la configuración.

Casi todas estas funciones estarán en lenguaje **SQL** y **Jinja**: SQL para la parte estática correspondiente a las consultas, y Jinja que será el lenguaje de templating que usaremos para poder crear funciones dinámicas reutilizables en todos los modelos que generarán las tablas finales.

La única función que no estará en uno de estos lenguajes será la relacionada con el archivo de configuración, ya que los elementos que usa DBT para esta función están en lenguaje **.YML**.

3.1.3. PowerBI

Esta será la última aplicación del proceso, la cual se encargará de mostrar la información recogida previamente por DBT y materializada como tablas en Snowflake.

Lo que tendremos que hacer será conectar PowerBI con Snowflake para poder acceder tanto a las tablas finales como a las tablas de origen, y diseñar una serie de **Dashboards** que muestren la información que hemos almacenado de la forma más clara y concisa posible.

3.2. Tipos de métricas

Antes de relacionar cada una de las funciones desarrolladas en el anterior apartado, es fundamental que pensemos bien qué métricas deben ser capaz de extraer nuestro framework.

Para esta labor, podemos hacer una separación en tres tipos de métricas: **Métricas de tabla, métricas de columna y métricas de campo.**

3.2.1. Métricas de tabla

Estas serán las menos numerosas, y se corresponden con las métricas que no tienen que ver con registros o conjuntos de registros específicos, sino que son propias de toda la tabla. Estas métricas se deberán configurar aparte de las otras en el archivo de parametrización, puesto que no es necesario aplicarles unas columnas objetivo, sino que se aplican sobre la tabla directamente. Las métricas de tabla que analizaremos serán las siguientes:

- **VOLUME**: Cantidad total de registros que hay en la tabla

- **SCHEMA_HASH**: Schema pasado por un algoritmo de hashing en el cual se recoge la información de restricciones, nombres de columnas, o tipado asociadas a la tabla.
- **LAST_UPDATED**: Fecha de la última actualización de la tabla

Como podemos observar, en estas métricas recogemos tres de los cuatro pilares de Data Observability que mencionamos que íbamos a analizar, faltando solo el pilar de Distribution, el cual estará asociado a todas las métricas recogidas en los otros dos tipos de métricas que vamos a introducir a continuación.

3.2.2. Métricas de columna

Tal como mencionamos antes, estas métricas están asociadas a Distribution, y abarcan valores estadísticos que nos dan información numérica sobre los datos que tenemos guardados. Estas métricas en concreto se aplican sobre conjuntos completos de columnas, y extraen información analizando todos los registros asociados a dicha columna. Las métricas que se analizan son:

- **NULL_RATE**: Relación entre 0 y 1 de la cantidad de nulos en nuestra columna contra la cantidad de registros no nulos.
- **UNIQUENESS**: Cantidad de registros únicos contra registros repetidos.
- **MINIMUM**: Mínimo registro dentro de la columna analizada.
- **MAXIMUM**: Máximo registro dentro de la columna analizada.
- **AVERAGE**: Media aritmética de todos los registros de la columna.
- **STANDART_DEV_POP**: Desviación media sobre toda la población de registros.
- **STANDART_DEV_SAMP**: Desviación media sobre una muestra considerada de los registros.
- **MED**: Mediana de todos los registros.
- **MAD**: Desviación absoluta de la mediana, analiza cuánto varía la mediana de un registro a otro.

3.2.3. Métricas de campo

Esta métrica será la encargada de analizar cada registro de una columna individualmente, comparándolo contra todos los demás campos de esa misma columna. Las cuatro métricas que analizaremos las siguientes:

- **MOVING_AVERAGE**: Media de un campo sobre un marco de tiempo especificado. Requiere un campo de fecha asociado y el marco de tiempo como parámetros.
- **Z_SCORE**: Medida estadística que analiza la aleatoriedad del registro con respecto al resto de registros de la columna.
- **MODIFIED_Z_SCORE**: Similar al Z score, pero en lugar de analizar la posibilidad de aleatoriedad, analiza cuánto difiere dicho registro del conjunto.
- **QUARTILE**: Cuartiles en los que se divide el conjunto de datos. Por defecto se divide en cuatro cuartiles, pero se puede variar este valor mediante un parámetro.

Como habremos podido notar, casi todos las métricas están asociadas a campos numéricos, que son a los que podemos aplicarles medidas estadísticas para el pilar de distribution. Algunas de estas métricas como el NULL_RATE, el MAXIMUM o el MINIMUM se pueden aplicar también a campos no numéricos, por lo que queda en mano del usuario el parametrizar apropiadamente el archivo de configuración para que se apliquen métricas numéricas solo en las columnas donde sea posible.

3.3. Flujo de datos

Ahora que entendemos de forma más específica qué función debe desempeñar cada una de las herramientas que participan en nuestro framework y qué tipos de métricas existen, vemos claramente que la mayor parte de nuestros esfuerzos deberán estar centrados en DBT, y en menor medida, en PowerBI.

El poder identificar qué vamos a tener que hacer en cada uno de los programas de forma aislada es muy importante, pero el diseño que hemos creado hasta ahora solo abarca funciones finales, pero no el flujo de los datos, por lo que ahora nos dedicaremos a tomar esas funciones, y pensar cómo podemos conectarlas unas con otras para llegar a cumplimentar nuestros objetivos.

Para realizar esta tarea, hemos planteado una división de los procesos de la aplicación en **tres capas** separadas, las cuales pueden incluir funciones de distintas aplicaciones, pero que tienen estrecha correlación entre ellas. Vamos ahora a desarrollar cada una de estas en mayor profundidad, así como a mostrar un pequeño diagrama que nos ayude a verlas entenderlas.

3.3.1. Primera capa

En la primera capa intervendrán **Snowflake y DBT**, y abarcará todo el proceso de análisis de datos de origen en Snowflake, y la extracción y materialización de las métricas asociadas a dichos datos de origen.

El proceso comenzará por la creación del **archivo de configuración**, donde se indicarán todas las tablas a las que queremos aplicar observabilidad, así como que tipos de métricas queremos extraerles, junto con algunas configuraciones más específicas. Para realizar esta tarea, usaremos el archivo **dbt_project.yml**, el cual es el archivo de parametrización genérico de todo proyecto de DBT.

Dentro de dicho archivo tenemos un apartado de variables, en el cual podemos generar estructuras **yaml** tan grandes o profundas como necesitemos. Estas variables se declaran a nivel global del proyecto, por lo que toda la información que plasmaremos aquí, podrá ser accedida posteriormente desde los modelos.

Estas estructuras yaml pueden ser accedidas e iteradas mediante Jinja, por lo que podrían ser directamente accedidas mediante las macros o los modelos, pero por cuestiones de estructuración, haremos que el que itere por este archivo y prepare los parámetros pertinentes para que las macros funcionen sea el **script de python**.

La idea de este script es que al ejecutarlo, itere por la estructura .yaml y que vaya accediendo a cada una de las tablas que analizaremos y creando y escribiendo los modelos pertinentes para llamar a las macros que se encargarán de realizar las operaciones especificadas en la configuración.

En caso de que estuviésemos usando una versión local de DBT levantada en un contenedor de docker, podríamos tener una solución más elegante en este punto, ya que esa versión permite la ejecución de scripts de python de forma local, y además guarda los proyectos localmente; pero estamos usando la versión cloud, la cual no guarda los archivos del proyecto de forma local, sino que lo gestiona todo en repositorios en la nube, y además no permite la ejecución de scripts, por lo que para realizar este paso tendremos que descargarnos localmente el proyecto desde el repositorio donde estemos guardándolo (en nuestro caso será **Azure Data Factory**), ejecutar el script localmente en el proyecto descargado, y luego resubir, o bien el proyecto entero, o los modelos nuevos que se han generado.

Por cada tabla que queramos analizar se crearán tres modelos, uno por cada uno de los tipos de métricas que hemos indicado previamente, y el contenido de estos modelos serán unas líneas de configuración que variarán el cómo se creará la tabla de destino en Snowflake, y una llamada a la macro correspondiente junto con toda la configuración necesaria para extraer las métricas deseadas.

Estas macros serán las encargadas de llevar todo el código SQL asociado a cada tipo de métrica para poder extraerlas, y también poseerán la lógica necesaria para seleccionar sólo las métricas solicitadas, sobre las columnas pertinentes, y con la parametrización especificada en caso de que fuera necesario. También deberán tener la capacidad de iterar por todas las columnas necesarias para poder ser capaz de adaptarse a la configuración sin tener unas macros con un código excesivamente denso.

Esta será la parte más desafiante del proyecto, ya que el crear las macros de esta forma, implica que tendrán que funcionar con parámetros de entrada extensos y complejos, por lo que tendremos que ser meticulosos con cada uno de los parámetros que incluyamos en el archivo de configuración para tratar de no complicar este proceso más de lo necesario.

Una vez todo este proceso haya terminado, tendremos lista para ejecutar la primera capa, necesitando ahora que el usuario ejecute mediante la línea de comandos la instrucción **dbt run**, ejecutando todos los modelos del proyecto, y generando así todos las tablas con las métricas recogidas en Snowflake.

En el siguiente esquema podemos ver cómo van a interactuar cada una de las distintas partes en la primera capa. También hemos añadido una parte de metadatos asociada al proyecto paralelo de extracción de linaje, pero esto es debido a que al hacer el dbt run, se ejecutará también esta parte, pero no es relevante para nuestro proyecto, ya que no realizaremos ninguna modificación a esa parte.

Destacamos que más adelante nos referiremos a las tablas resultantes de esta capa como **tablas de métricas.**

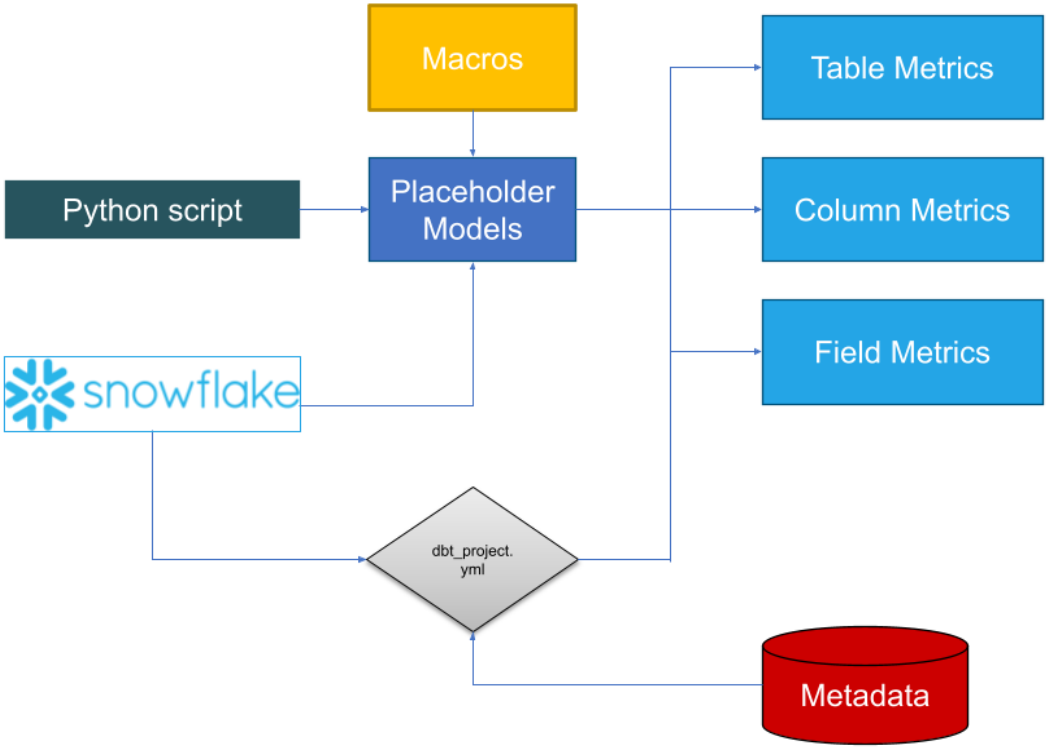

Ilustración 5 Capa 1

3.3.2. Segunda capa

La primera capa es la encargada de generar todas las métricas que nos van a servir para identificar si los datos que tenemos guardados están en buen estado, o muestran algún tipo de particularidad que pueda ser indicativa de algún tipo de problema, pero no tienen ninguna funcionalidad que permita analizar esas particularidades o desviaciones, por lo que solo con esas tablas, un usuario tendría que acceder manualmente a todos los registros y comprobarlos para ver si encuentra alguna discrepancia.

Es aquí donde la segunda capa da solución a este problema. En esta capa encontraremos todas las funciones que analicen las métricas creadas previamente, y que sean capaces de indicar si algunas de estas presentan algún indicativo de que los datos están en mal estado.

Las aplicaciones que intervendrán serán también **Snowflake** y **DBT**, y el mecanismo que usaremos para el análisis de las métricas será uno sencillo, que se base solo en un valor máximo o mínimo proporcionado por el usuario a modo de **umbral**, y que el sistema reconozca los registros que superen dicho umbral como **registros erróneos**.

Destacamos que en un primer planteamiento tratamos de plantear un sistema más complejo, que usase datos relativos a los registros de la propia columna, pero aumentaba considerablemente la dificultad y el consumo de tiempo, cosa que no podemos permitirnos teniendo en cuenta la planificación y el tiempo total del que disponemos para finalizar el proyecto.

El funcionamiento de esta capa es similar al de la primera, ya que reutiliza algunos de los recursos de esta. Para empezar, los umbrales que el usuario deberá aplicar están también definidos en el archivo de configuración, asociados a cada una de las tablas y columnas que haya configuradas, permitiendo así asignar un mismo umbral a distintas métricas y columnas, minimizando la configuración que el usuario debe escribir.

Esta configuración se lee también mediante el mismo script de python de la primera capa, y crea otra serie de modelos con una dinámica similar a los de la anterior capa, pero llamando a otras macros algo más sencillas, que se encargan de acceder a los modelos creados en esa capa, y aplicarles los umbrales que el usuario ha solicitado.

Por cada registro que estas macros reconocen como erróneo, se añade un registro a una tabla nueva, incluyendo el valor de origen, la métrica analizada, el umbral, y cuánto ha diferido del umbral especificado.

El resultado será una serie de tablas (tres o menos, ya que no se crearán tablas de más si no se ha aplicado ningún umbral de un tipo de métrica concreta), las cuales tienen todos los registros que han excedido los umbrales del usuario, dándole una forma sencilla de identificar patrones comunes y de poder analizar individualmente todos los registros que no se ajustan a los umbrales que deseaba.

Destacamos que los modelos que genera el script de python toman como parámetros algunos de los específicos de la primera capa, por lo que debemos respetar la estructura que hayamos definido en la primera capa para ejecutar la segunda.

En caso de que queramos realizar alguna modificación en los parámetros de la segunda capa, será necesario ejecutar el script de python nuevamente, y ejecutar la primera y luego la segunda, pero no será necesario ejecutar el script dos veces para cada capa, ya que los modelos de ambas se generan simultáneamente.

Llamaremos posteriormente a las tablas generadas en esta capa: **Tablas de análisis.** La ilustración 6 contiene una representación gráfica de esta capa.

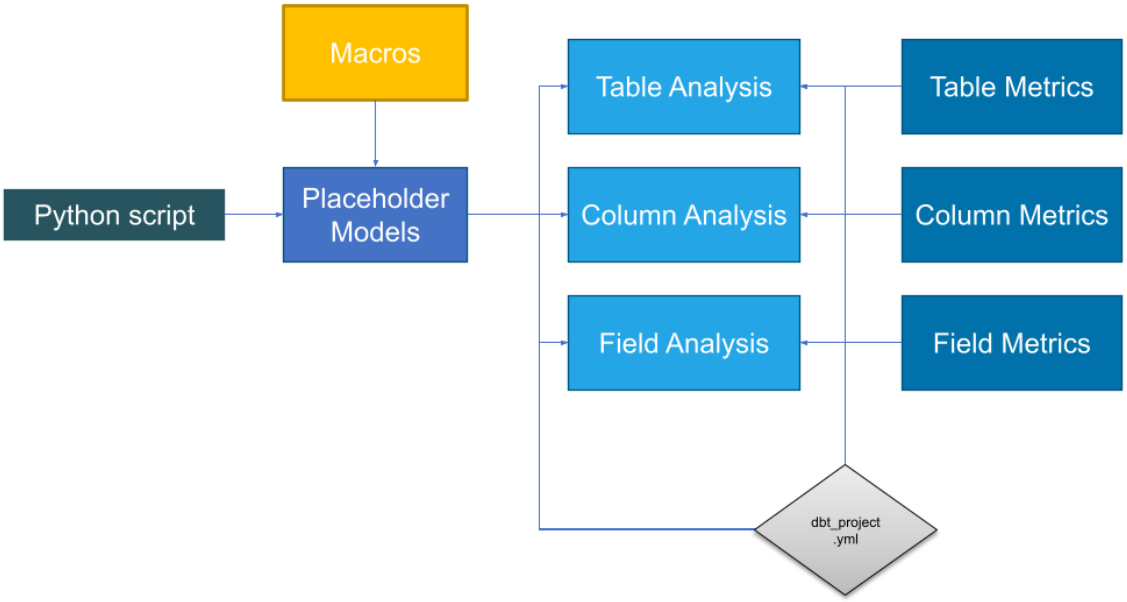

Ilustración 6 Capa 2

3.3.3. Tercera capa

Esta será la última capa de nuestro framework, y en ella no intervendrá DBT, solo intervendrán **Snowflake** y **PowerBI**.

Esta capa abarcará la recepción de la información de Snowflake en el **dashboard final**, y la plasmación de la misma en las distintas páginas de este. La idea será tomar tanto las tablas de **métricas** como las de **análisis** para poder compararlas en distintos niveles de granularidad.

El **primer nivel** mostrará una **vista de pájaro** de todas las tablas, junto con sus métricas, indicando cuáles de ellas tienen registros erróneos así como el porcentaje de registros erróneos respecto del total de registros de cada tabla. Tendremos también un indicativo visual con colores en función de este porcentaje, indicandonos como de acuciante podría ser el problema en cada tabla, marcando en verde las tablas sin registros erroneos, en amarillo las que tienen errores pero son menos de un 20%, y en rojo las que superan el 20%.

En el **segundo nivel**, bajaremos un nivel de granularidad, dedicando una página del dashboard por cada tabla de origen que hayamos analizado, mostrando en esta sus tres tipos de tablas de métricas y comparándolas contra sus tablas de análisis, mostrando los registros totales contra los registros erróneos con mayor detalle.

Finalmente, el **tercer nivel** será el último, donde en cada página analizaremos individualmente cada métrica asociada a una tabla. Las anteriores tablas comparan sólo los registros de la última ejecución, pero en este nivel veremos un histórico de la métrica en cuestión, indicando cómo ha ido cambiando a lo largo del tiempo para poder ver su progresión.

Destacamos que la creación de las páginas es un paso manual, ya que PowerBI no permite la creación de páginas dinámicas por cada registro, por lo que contamos con que el usuario creará solo páginas para las métricas que más le interesen en ese momento, y no creará una para cada una de las métricas que tenga registradas.

Encontramos en la ilustración 7 una representación gráfica de esta capa:

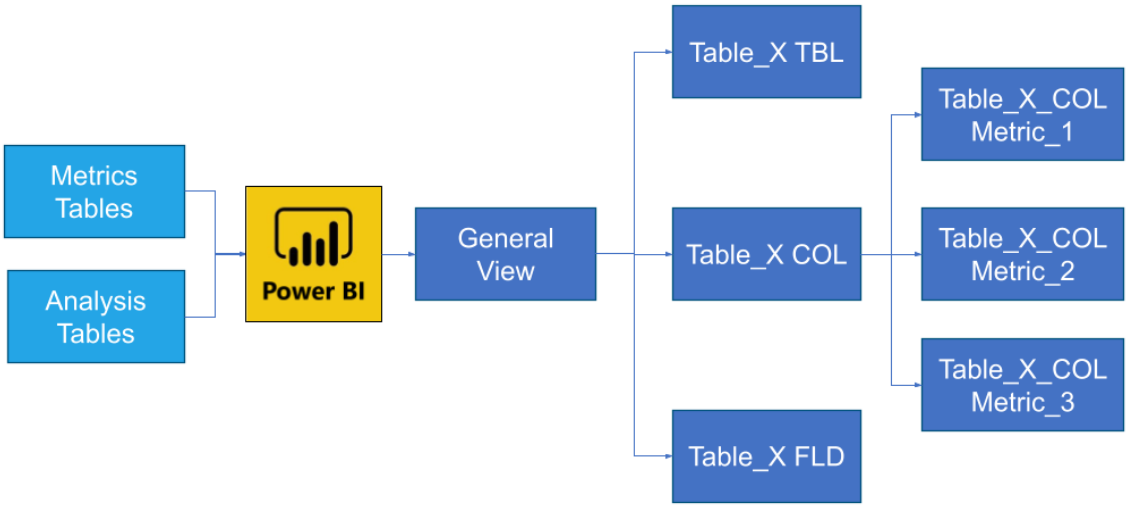

Ilustración 7 Capa 3

3.4. Conclusiones del diseño

La fase de diseño ha transcurrido satisfactoriamente, dejando un mapa de ruta a seguir claro y que se adecua apropiadamente tanto a la planificación del proyecto como a las capacidades personales que poseemos para llevar adelante el trabajo.

No obstante, debemos tener en cuenta que del mismo modo que pueden llegar a haber desviaciones en la planificación del proyecto, las puede haber también en el diseño: Contratiempos que no esperábamos o dificultades inesperadas en la implementación nos pueden llevar a tener que alterar algunos de los puntos clave del diseño del framework, por lo que, dado nuestro tiempo limitado, en algunas de estas situaciones asumimos que será mejor para el proyecto desviarnos lo necesario del diseño original en pos de perder el mínimo tiempo posible con estos contratiempos.

Este tipo de problemas que probablemente acaben por surgir, estarán recogidos en la sección posterior, la de implementación, donde hablaremos de los contratiempos que no pudimos prever con antelación, y qué decisiones nos vimos obligados a tomar para solventarlos.

4. IMPLEMENTACIÓN

Ya completadas todas las fases relacionadas con planificación, análisis y diseño, podemos finalmente pasar a la parte más importante del proyecto: La implementación del framework que hemos diseñado.

Será en esta fase donde trataremos de aplicar toda la información que hemos recogido y todos los procesos que hemos planificado para crear el propio framework. A nivel de la memoria, nos centraremos principalmente en recoger cómo efectuamos a bajo nivel cada una de las tareas necesarias para cumplimentar el diseño que hemos preparado. Trataremos de recoger dichas tareas en el mismo orden en el que las desarrollamos, y reservaremos una subsección al final para comentar los problemas que encontramos y qué solución aplicamos para sortear cada uno de ellos.

Para facilitar la redacción de esta sección, la dividiremos en distintas partes, basándonos para ello en el diseño de tres capas que mencionamos previamente, y añadiendo un apartado más para las tareas previas al desarrollo del código.

4.1. Preparación de las aplicaciones

Antes de saltar directamente a generar código, deberemos realizar las configuraciones necesarias para que nuestras aplicaciones funcionen e interactúen entre ellas como necesitamos.

4.1.1. Creación de la cuenta

Lo primero será crear una cuenta gratuita de Snowflake, lo cual es bastante sencillo, solo tenemos que introducir nuestros datos y obtendremos una cuenta con un tiempo de uso de unos dos meses. En un proyecto más grande convendría pagar una suscripción para no tener límite de tiempo ni de capacidad de computación, pero en este proyecto no tendremos problemas usando la cuenta gratuita.

Una vez creada la cuenta, nos encontraremos las dos bases de datos que vienen ya creadas por defecto: "**SNOWFLAKE**", donde se recogen en tablas la información relacionada con la cuenta, y "**SNOWFLAKE_SAMPLE_DATA**", donde tenemos las tablas con información genérica que usaremos para nuestros ejemplos, más concretamente utilizaremos la tabla "**CUSTOMERS**" donde se recoge información sobre ciento cincuenta mil clientes ficticios.

C_CUSTKEY	C_NAME	C_ADDRESS	C_NATIONKEY	C_PHONE	C_ACCTBAL
60,001	Customer#000060001	9li4zQn9cX	14	24-678-784-9652	9,957.56
60,002	Customer#000060002	ThGBMjDwKzkoOxhz	15	25-782-500-8435	742.46
60,003	Customer#000060003	Ed hbPtTXMTAsgGhCr4HuTzK,Md2	16	26-859-847-7640	2,526.92
60,004	Customer#000060004	NivCT2RVaavl,yUnKwBjDyMvB42WayXCnky	10	20-573-674-7999	7,975.22
60,005	Customer#000060005	1F3KM3ccEXEtl, B22XmCMOWJMl	12	22-741-208-1316	2,504.74
60,006	Customer#000060006	3isiXW651fa8p	22	32-618-195-8029	9,051.4
60,007	Customer#000060007	sp6KJmx,TiSWbMPvhkQwFwTuhSi4a5OLNlmpcGl	12	22-491-919-9470	6,017.17
60,008	Customer#000060008	3VteHZYOfbgQioA96tUeL0R7i	2	12-693-562-7122	5,621.44

Ilustración 8 Tabla origen CUSTOMERS

Una vez hemos localizado estas tablas, lo que deberemos hacer será crear ahora nuestra propia base de datos, que llamaremos "**DATA_OBS**", y será aquí donde guardaremos toda la información que extraigamos de las tablas de origen. No será necesario crear ningún esquema (agrupaciones de tablas dentro de una base de datos) para dichas tablas ya que esto lo gestionaremos a nivel de DBT, pero sí que tendremos que crear uno para subir los archivos necesarios para que el proyecto paralelo de extracción de datos de linaje funcione.

4.1.2. Archivos del proyecto paralelo

Estos archivos serán un conjunto de tablas que se usan a modo de tablas fuente en dicho proyecto.

Subir estas tablas es más complicado de lo que podría parecer en primer lugar, ya que Snowflake no tiene en su versión web ningún método para subir archivos locales, es por ello que

deberemos descargar e instalar **Snowsql** en una consola local, una librería que permite, entre otras cosas, cargar archivos locales en un **Stage** de una cuenta de Snowflake, para más adelante poder extraer estos archivos en la web de Snowflake.

El proceso de carga de archivos en un stage es bastante simple, solo necesitamos introducir los credenciales de nuestra cuenta de Snowflake y usar el comando **put [ruta] @[nombre_del_stage]**, y una vez en Snowflake, podremos usar en una consulta el comando **copy into** para pasar los archivos de un stage a una tabla.

Este proceso en sí mismo no es complicado, pero hay una dificultad bastante relevante, y es que las tablas que tenemos que subir a Snowflake tienen en algunas de sus celdas archivos .json, los cuales Snowflake tiene la capacidad de leer, pero no tiene la capacidad de cargar, no como un archivo en una celda. Esto desencadena que por cada una de las tablas que tenemos que subir, debemos separar manualmente el contenido de esas celdas y subirlo individualmente a una tabla con una sola columna usando la opción de configuración del comando copy into para archivos json: **copy into json_adf from @MY_UNLOAD_STAGE/Json1Prueba.csv.gz file_format=(TYPE=json record_delimiter='@@' field_delimiter='%%' skip_blank_lines=true)**

Posteriormente al tener los archivos .json en una tabla, debemos modificar dichas tablas para añadirles las columnas que faltan y subir por separado el resto de campos que faltaban. Este proceso no es algo común, ya que el hecho de no poder subir tablas con celdas con archivos .json es un bug reconocido de Snowflake, y el procedimiento que llevamos a cabo es tan solo una solución que nosotros hemos ideado para poder sortear este problema.

4.1.3. Configuración de DBT

Ya preparada la cuenta de Snowflake, ahora pasamos a vincularla con DBT. Para hacer esto, debemos crear un proyecto nuevo en DBT Cloud, y crear una conexión con nuestra base de datos. Los parámetros que le proporcionamos serán: El enlace de nuestra cuenta de Snowflake (cada cuenta tiene un enlace de login único asociado), nuestro usuario, nuestra contraseña y la base de datos con la que interactuaremos, en este caso será DATA_OBS.

Con esto DBT debería ser capaz de extraer y subir información a nuestra base de datos para así poder realizar los análisis y subir los resultados.

Otro paso fundamental en la configuración de DBT es preparar el almacenamiento del código. DBT trabaja de forma nativa usando la metodología **GIT** para el control de versiones, por lo que necesitaremos un repositorio externo para esta labor. En nuestro caso, SDGGroup habitualmente trabaja con repositorios de **Azure DevOps**, y nos ha proporcionado uno para que subamos el código del proyecto a este, por lo que el siguiente paso será configurar el repositorio.

DevOps trabaja con claves SSH para identificar y autenticar tanto los usuarios como los proyectos, por lo que para conseguir vincular ambos programas, debemos generar una clave SSH mediante la opción exportar de DevOps, y pegar esa clave en la configuración de nuestro proyecto DBT.

Ya completados estos pasos, añadiremos alguna configuración extra para facilitarnos nuestra tarea, como especificar un esquema por defecto para la creación de las tablas, para que así no se mezclen con las tablas fuente del proyecto paralelo. Siguiendo la metodología habitual de la empresa, llamaremos al esquema **dbt_rbenito.**

Finalmente, en la carpeta “models”, donde habitualmente se crean todos los modelos del proyecto, encontraremos un archivo llamado **schema.yml**, donde declararemos todas las tablas de origen de donde extraeremos las métricas, y les asignaremos una agrupación para

referenciarlas en DBT. En este caso, asignamos a la tabla orders la agrupación **SAMPLE_OBSERVABILITY**.

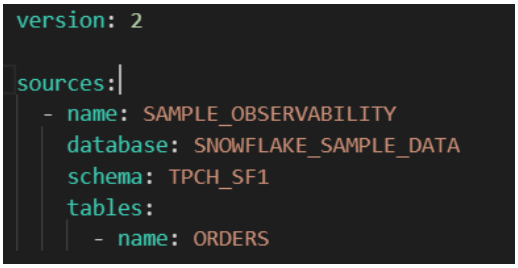

Ilustración 9 Declaración de fuentes

4.1.4. Configuración de PowerBI

También debemos preparar nuestra herramienta visual para que reciba los datos que vayamos generando, para ello de momento nos limitaremos a configurar Snowflake como base de datos de origen, pero todavía no vamos a indicar los nombres de las tablas.

El motivo de esto es que PowerBI es el último eslabón en nuestro framework, por lo que en caso de que haya cambios en el diseño, este es el que más probabilidades tiene de verse afectado, por lo que por ahora, simplemente tendremos la base de datos conectada, y cuando sepamos a ciencia cierta exactamente qué tablas vamos a utilizar, volveremos a la configuración de este programa y le pasaremos los nombres y esquemas de estas tablas para que se conecte a ellas y pueda extraer los campos necesarios.

4.2. Primera capa

Ahora que tenemos configuradas las plataformas que serán la base para nuestro framework, es el momento de comenzar a desarrollar el código que ejecutará cada una de las funciones que hemos detallado previamente. Vamos ahora a dividir las distintas etapas de la implementación por las capas que establecimos previamente, empezando por la primera, la cual, recordemos que abarca desde el punto de partida del proceso, hasta la materialización como tablas nuevas de las métricas asociadas a cada una de las tablas que hayamos indicado que queremos observar en el archivo **dbt_project.yml.**

4.2.1. Dbt_project.yml

Empezaremos por el archivo que contendrá toda la configuración del proyecto que puede suministrar el usuario objetivo.

El dbt_project.yml es un archivo genérico que se genera para cada proyecto de DBT. En dicho archivo se declaran multitud de propiedades del proyecto que alteran la forma de funcionar de DBT, pero adicionalmente, tiene un apartado de variables donde podemos declarar cualquier cosa siempre que sea en formato **.yaml**.

La forma de funcionar de los formatos .yaml hace que podamos generar conjuntos de variables ramificadas en tantas categorías como necesitemos, lo cual lo hace muy conveniente para

declarar grandes cantidades de variables relacionadas las unas con las otras, como es nuestro caso.

La idea será declarar una única variable, que sea un árbol .yaml muy profundo y amplio, donde esté toda la configuración que el usuario debe declarar, de modo que no necesite ninguna interacción adicional para hacer funcionar el framework, más allá de usar los comandos para ejecutar los modelos.

El árbol .yaml comienza por las propiedades comunes para el framework, y va ramificándose a medida que bajamos el nivel de granularidad de la configuración. La primera parte hace referencia al sufijo que se añade a los nombres de las tablas resultantes de la capa uno y dos. El usuario puede elegir el sufijo o puede dejar los que vienen por defecto.

```yaml
observability:
  config:
    metrics_table_suffix: '__obs' #Optional, default is '__obs'
    analysis_table_suffix: '__anly' #Optional, default is '__anly'
```

Ilustración 10 Sufijos

La siguiente ramificación hace referencia tanto a la tabla objetivo como a la configuración de las métricas de tabla de la misma. La referencia a la tabla viene dada por la propiedad "**object**", donde si estamos apuntando a una tabla fuente de Snowflake, declararemos un array con el primer elemento indicando el nombre de la agrupación que declaramos en el **schema.yml**, y con el nombre de la tabla como segundo parámetro (ejemplo de este caso en la ilustración 12).

En caso de que queramos extraer métricas de un modelo de DBT en vez de una tabla de Snowflake, podemos hacerlo declarando la propiedad object como un String que contenga el nombre del modelo objetivo: **- object: 'modelo_de_ejemplo'**

Tal como mencionamos, también encontramos en este nivel la configuración de las métricas de tabla, bajo el nombre de **metrics**. Permiten al usuario introducir un array con los nombres de las métricas que desea aplicar, y le dan la posibilidad de dejarlo vacío si quiere aplicar todas.

También encontramos los llamados **thresholds**, que indican cuáles son los límites superiores o inferiores que el usuario quiere aplicar sobre cada métrica, para que, en caso de ser rebasados, sean reconocidos como registros problemáticos en la segunda capa.

Finalmente, en este nivel encontramos el parámetro **meta**, donde podemos introducir configuración genérica para la tabla necesaria para hacer funcionar algunas de las métricas, por ejemplo: Debemos introducir el nombre de una columna que haga referencia a fecha para poder calcular la **moving average**.

```yaml
- object: ['SAMPLE_OBSERVABILITY','ORDERS'] #Array or String, depending on the type of source
  metrics: [] #Table metrics, column and field are configured below
  thresholds:
    - metric: 'VOLUME'
      upper_limit: 100
      lower_limit: 10
  meta:
    timestamp_column: 'O_ORDERDATE' #Target column for date in moving average.
```

Ilustración 11 Métricas de tabla

Bajando al siguiente nivel, nos encontramos el apartado **columns** donde, siguiendo una estructura de sets que nos permitan declarar tantos sets de columnas como necesitemos, podremos aplicar sobre las columnas que queramos las métricas que especifiquemos, dándonos también la posibilidad de introducir thresholds específicos en función de la columna y la métrica.

En este caso declararemos a la vez las métricas de columna y de campo, y al igual que con las de tabla, podemos dejar el array correspondiente vacío para aplicar todas las métricas. Adicionalmente, tenemos dos campos extra llamados **exclude_metrics** y **exclude_columns**, donde podemos introducir métricas y columnas respectivamente que no queramos aplicar en este set, evitando que el usuario tenga que declarar todas las métricas o columnas manualmente en caso de que quiera analizar todo descontando unas pocas columnas o métricas en específico.

```yaml
columns:
#Las columnas de datos no numéricos solo debe aplicarseles NULL_RATE y UNIQUENESS.
  - column_set:
    columns: [O_TOTALPRICE,O_CUSTKEY] #Names of the columns to analize. Default all.
    exclude_columns: [] #Names of the columns to exclude.
    metrics: [] #Names of the metrics to apply. Default all.
    exclude_metrics: ['AVERAGE','MOVING_AVERAGE'] #Names of the metrics to exclude.
    thresholds:
    - metric: 'MINIMUM' #Metric to apply the thresholds
      column_set: #Set of columns on this specific metric
      - columns: [O_TOTALPRICE,O_CUSTKEY] #Columns to apply thresholds in the MINIMUM metric
        upper_limit: 1000
        lower_limit: 100
    - metric: 'MAXIMUM'
      column_set:
      - columns: [O_TOTALPRICE,O_CUSTKEY]
        upper_limit: 2000
        lower_limit: 200
```

Ilustración 12 Sets de columnas

Para terminar, dentro de un mismo set de columnas tenemos un apartado de **config**, donde podemos indicar parámetros para el cálculo de algunas métricas específicas. En nuestro caso, podemos aplicar estos parámetros al intervalo de los días que habrá para el cálculo del moving_average, y las porciones en las que dividiremos los registros para la métrica de cuartiles.

```yaml
    config:
      moving_average_days: '50' #Moving averange days interval
      quartile_segments: '6' #Portions to calculate quartiles
```

Ilustración 13 Configuración de métricas

Destacamos que en un principio estuvimos planteando declarar toda esta configuración en un archivo .yml aparte del dbt_project para no tener la configuración de nuestro framework mezclada con los otros parámetros usados para configurar a nivel de proyecto. Acabamos por descartar esta idea puesto que, tras revisar cómo funcionaban otros paquetes de macros oficiales de DBT, nos dimos cuenta de que todos declaraban los parámetros que necesitaban en el apartado de variables del dbt_project.

4.2.2. Macros

Ya creado el archivo que contiene toda la configuración del framework, vamos ahora a crear los archivos que contendrán toda la lógica de la extracción de métricas: las macros.

DBT proporciona estos archivos como una aproximación a las funciones de otros lenguajes de programación, donde podemos definir un fragmento de código que actúa en base a unas variables que se proporcionan como parámetros de entrada, y que luego podemos llamar en los modelos de DBT, ahorrando espacio y haciendo más dinámico el proceso.

Necesitaremos crear tres macros genéricas, una por cada tipo de métrica, en ellas se generará todo el código SQL que luego se inyectará y ejecutará en Snowflake para crear las tablas de métricas pertinentes.

Empezaremos por la macro **get_table_metrics**, que será la encargada de extraer las métricas de tabla. Para empezar, esta macro tiene una llamada a una función de DBT llamada **adapter.dispatch**: una macro nativa que, dada una función proporcionada como parámetro, DBT discierne a qué base de datos estamos conectados, y en función de cual sea llama a la función asociada solo a dicha base de datos.

```
{% macro get_table_metrics(metrics, thresholds ,source_table) -%}
    {{ return(adapter.dispatch('get_table_metrics')(metrics, thresholds, source_table)) }}
{%- endmacro %}
```

Ilustración 14 Selector de bases de datos

Este comportamiento permite tener distinto código asociado a una misma macro que varíe dependiendo de a qué base de datos nos conectamos, dando flexibilidad a la hora de desarrollar para distintas fuentes simultáneamente. En nuestro caso solo vamos a usar Snowflake, por lo que solo tenemos creada una función con código orientado a esa BD.

Vamos ahora a la declaración de la macro, donde definimos el nombre de la misma y los parámetros de entrada que necesitará para funcionar. En esta tendremos por nombre get_table_metrics y usaremos como parámetros un array llamado **metrics**, otro llamado **thresholds**, y un string con la referencia a la tabla de origen llamado **source_table**. Veremos en el posterior apartado de donde surgen cada uno de estos parámetros.

```
{% macro snowflake__get_table_metrics(metrics, thresholds, source_table) -%}
```

Ilustración 15 Cabecera de métricas de tabla

El siguiente paso será desarrollar un bloque de código por cada una de las métricas que vamos a extraer. Este bloque debe tener la capacidad de identificar según el array de metrics, qué métricas específicas queremos extraer de la tabla fuente; podemos hacer esto con las funciones **if** que nos proporciona Jinja, y combinarlo con algunas de las funciones básicas de SQL y otras que nos proporciona Snowflake para realizar el cálculo.

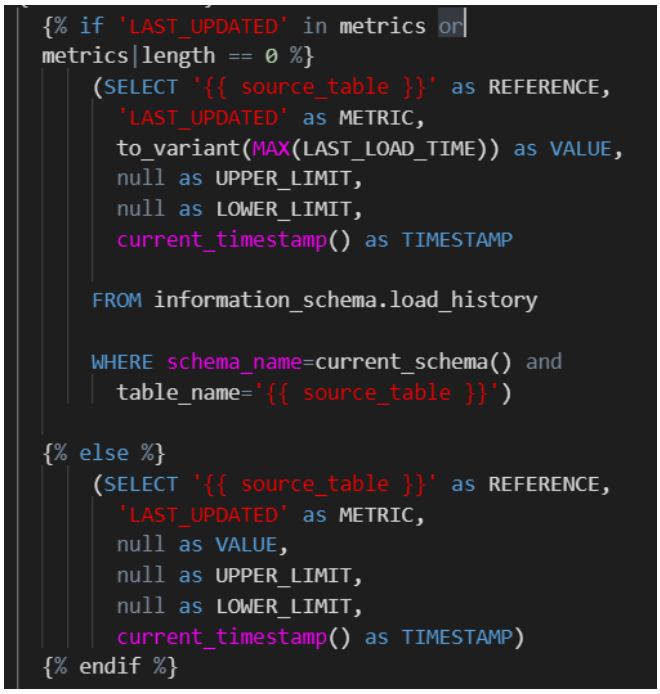

Ilustración 16 Consultas métricas de tabla

En la anterior ilustración, podemos apreciar cómo es la estructura para el cálculo de la métrica **LAST_UPDATED**, donde primero hay un if para comprobar si la métrica está entre las seleccionadas, y en caso de que lo esté, se hace una llamada al parámetro **LAST_LOAD_TIME** de la tabla **load_history**, una tabla genérica de Snowflake que contiene toda la información de las actualizaciones de las tablas que hay guardadas en la BD.

En caso de no querer analizar esta métrica, dejamos todos los campos que intervienen en el proceso a null, a excepción de **METRIC** y **TIMESTAMP**. Posteriormente, hacemos un **UNION** y seguimos un proceso similar con las otras dos métricas de tabla, pero variando la forma de calcular la columna **VALUE** en función de qué métrica estemos calculando.

Destacamos que se generará una fila en las tablas finales por cada métrica que analicemos, en el caso de esta tabla serán tres filas, y en el caso de las siguientes, dependerán de la cantidad de columnas y de campos de dichas tablas.

Pasamos ahora a la macro asociada a las métricas de columna, llamada **get_column_metrics**. Esta macro tiene una estructura inicial similar a la anterior, con la particularidad de que solo tiene como parámetros **column_sets**, y **source_table**. El segundo parámetro solo contiene la referencia a la tabla de origen, pero el primero contiene todos los sets de columnas asociados a esa tabla de origen en el dbt_project.yml.

```
{% macro snowflake__get_column_metrics(column_sets, source_table) -%}
```

Ilustración 17 Cabecera métricas de columnas

La lógica de la macro está dividida en dos partes: Una que itera por la estructura de **column_sets**, y que genera una serie de variables con las métricas a analizar, las columnas y la configuración; y una segunda parte donde se hace un bucle que itera por cada columna del set, y que tiene otro bucle dentro que itera sobre cada una de las métricas solicitadas. Todo el proceso está englobado en otro bucle que hace una iteración por cada set de columnas que tengamos declarado.

En esta primera parte, se realizan algunas comprobaciones para ver si el usuario desea analizar todas las métricas o solo algunas, y guarda las métricas a analizar en un array, mientras que posteriormente, se hace un bucle para retirar de dicho array las columnas que el usuario había indicado que quería excluir.

```
{% if set_metrics|length==0 %}
    {% set set_metrics=['NULL_RATE','UNIQUENESS','MINIMUM','MAXIMUM','AVERAGE','STANDART_DEV_POP','STANDART_DEV_SAMP','MED','MAD'] %}
{% elif ('NULL_RATE' not in set_metrics) and ('UNIQUENESS' not in set_metrics)
and ('MINIMUM' not in set_metrics) and ('MAXIMUM' not in set_metrics)
and ('AVERAGE' not in set_metrics) and ('STANDART_DEV_POP' not in set_metrics)
and ('STANDART_DEV_SAMP' not in set_metrics) and ('MED' not in set_metrics)
and ('MAD' not in set_metrics) %}
    {% set set_metrics=[] %}
    {% set set_columns=[] %}
{% endif %}
{% for exclude_metric in column_set['exclude_metrics'] %}
    {% if exclude_metric in set_metrics %}
        {% set temp=set_metrics.pop(set_metrics.index(exclude_metric))%}
    {% endif %}
{% endfor %}
```

Ilustración 18 Selección de métricas y columnas

Se aplica un proceso similar al descrito anteriormente para la exclusión de las columnas, y se pasa después a la parte de los bucles. En esta parte, por cada columna y por cada métrica, pasaremos por una serie de bloques **if**, donde: En caso de que la métrica del if sea una de las que queremos analizar, se crea un registro aplicando el cálculo pertinente a la métrica, y en caso contrario, se pasa al siguiente if, hasta que encontremos el correspondiente a la métrica.

```
{% for column_name in set_columns %}
    {% for check_metric in set_metrics %}
        (select '{{source_table}}' as REFERENCE,
        '{{column_name}}' as COLUMN_NAME,
        '{{check_metric}}' as METRIC_NAME,
        {% if 'NULL_RATE' == check_metric %}
            SUM(CASE WHEN {{column_name}} IS NULL THEN 1 ELSE 0 END) / COUNT(*) AS METRIC_VALUE,
        {% endif %}
        {% if 'UNIQUENESS' == check_metric %}
            COUNT(DISTINCT {{column_name}})/COUNT({{column_name}}) as METRIC_VALUE,
        {% endif %}
        {% if 'MINIMUM' == check_metric %}
            to_varchar(min({{column_name}})) as METRIC_VALUE,
        {% endif %}
        {% if 'MAXIMUM' == check_metric %}
            to_varchar(max({{column_name}})) as METRIC_VALUE,
        {% endif %}
```

Ilustración 19 Consultas métricas de columnas

Después de estos if para poblar la columna **value**, tenemos otro conjunto de if, esta vez para comprobar si tenemos **thresholds** o no. En caso de tenerlos y que se correspondan con la métrica y la columna con la que estamos trabajando, añadimos sus valores a los campos **LOWER_LIMIT** y **UPPER_LIMIT**, pero no hacemos ninguna comprobación en este nivel de si el value sobrepasa alguno de estos o no, ya que esto lo haremos en la segunda capa.

```
{% set nstemp=namespace(upper_limit = 'null',lower_limit = 'null') %}
{% if column_set['thresholds'] is not none and column_set['thresholds']|length!=0 %}
    {% for thresholds_metrics in column_set['thresholds'] %}
        {% if thresholds_metrics['metric'] == check_metric %}
            {% for thresholds_column_set in thresholds_metrics['column_set']  %}
                {% if column_name in thresholds_column_set['columns'] and thresholds_column_set['upper_limit'] != null %}
                    {% set nstemp.upper_limit= thresholds_column_set['upper_limit'] %}
                {% endif %}
                {% if column_name in thresholds_column_set['columns'] and thresholds_column_set['lower_limit'] != null %}
                    {% set nstemp.lower_limit= thresholds_column_set['lower_limit'] %}
                {% endif %}
            {% endfor %}
        {% endif %}
    {% endfor %}
{% endif %}
{{nstemp.upper_limit}} as UPPER_LIMIT,
{{nstemp.lower_limit}} as LOWER_LIMIT,
```

Ilustración 20 Aplicación de límites

Finalmente, hacemos un par de comprobaciones valiéndonos de la macro **loop.last** (nos indica si el elemento actual es el último del bucle) para cerrar apropiadamente los tres bucles for que hemos abierto.

```
            {% if not set_columns==[] and not loop.last %}
                UNION
            {% endif %}
        {% endfor %}
        {% if not set_columns==[] and not loop.last %}
            UNION
        {% endif %}
    {% endfor %}
    {% if not set_metrics==[] and not loop.last %}
        UNION
    {% endif %}
```

Ilustración 21 Cierres de los bucles

La última macro que tenemos en esta capa será la de métricas de campo: **get_field_metrics.** Posee los mismos parámetros de entrada que la de métricas de columna, y además posee un parámetro extra para el cálculo del **mean_average**: La columna de **timestamp**.

```
{% macro snowflake__get_field_metrics(column_sets, source_table, timestamp_column) -%}
```

Ilustración 22 Cabecera métricas de campo

La estructura que posee esta macro es muy similar a la de métricas de tabla, difiriendo sólo en la forma de calcular cada una de las métricas. Vemos en la siguiente ilustración unos ejemplos de los cálculos de **mean_average** y **quartile:**

```
(select '{{source_table}}' as REFERENCE,
'{{column_name}}' as COLUMN_NAME,
{{column_name}} as COLUMN_VALUE,
{% if 'MOVING_AVERAGE' == check_metric %}
    'MOVING_AVERAGE_{{column_set['config']['moving_average_days']}}' as METRIC_NAME,
    avg({{column_name}})
        over (order by {{timestamp_column}} rows between {{column_set['config']['moving_average_days']}} preceding and current row)
    as METRIC_VALUE,
{% endif %}
{% if 'QUARTILE' == check_metric %}
    'QUARTILE_{{column_set['config']['quartile_segments']}}' as METRIC_NAME,
    ntile({{column_set['config']['quartile_segments']}}) over (partition by {{column_name}} order by {{column_name}})
    as METRIC_VALUE,
{% endif %}
```

Ilustración 23 Consulta métricas de campo

Destacamos que la tabla generada por esta macro será la más extensa, ya que genera un registro por cada campo individual, por cada columna y por cada métrica, por lo que una tabla con 2 columnas, 10 registros y que queramos aplicarle 5 métricas, tendrá como resultado 100 registros. Por suerte, Snowflake está preparado para almacenar millones de registros, por lo que estas cifras no nos supondrán problemas de almacenamiento.

4.2.3. Script de python

El siguiente paso del framework es realizar las llamadas a las macros consumiendo la configuración del archivo dbt_project. DBT posee un comando para extraer variables de dicho archivo por lo que podemos crear un modelo que haga una llamada a dicha variable, y que la pase como parámetro a la macro correspondiente, no obstante, por petición del cliente y para aislar el resto de configuraciones que hay en el dbt_project.yml, vamos a evitar usar dicha variable.

Por lo expuesto anteriormente, necesitaremos una alternativa para acceder a la configuración y plasmarla en los modelos que posteriormente llamarán a cada una de las macros. La alternativa que usaremos será un **script Python**, el cual, accediendo al dbt_project.yml, extraiga la configuración de este y genere todos los modelos necesarios para la primera capa con el nombre apropiado (ya que el nombre del modelo define el nombre de la tabla en la que se materializa en Snowflake) y con el código pertinente.

El script deberá leer el archivo dbt_project.yml y posteriormente, identificar cuántos modelos serán necesarios, y hacer un bucle que genere cada uno de ellos. Como vamos a tener varias partes con código muy similar y vamos a estar trabajando principalmente con strings, vamos a crear variables en el script para el código común, y que tengan parámetros que sean las partes que variarán de un fragmento a otro.

```
# Literal
OBSERVABILITY_PATH = 'models/observability_models/'
ANALYSIS_PATH = 'models/observability_analysis/'

# Table obs. model
HEADER_METRIC = "{{ config(materialized=\'incremental\', schema=\'observability_tables\') }}\n"
SOURCE_L = '{{%-set source_table = source("{src1}","{src2}")-%}}\n'
SOURCE_S = '{{%-set source_table = ref("{model}")-%}}\n'
TABLE_METRIC = '{{{{get_table_metrics({metrics}, {thresholds}, source_table)}}}}'
COLUMN_METRIC = '{{{{get_column_metrics({columns}, source_table)}}}}'
FIELD_METRIC = '{{{{get_field_metrics({columns}, source_table,"{meta_time}")}}}}'

HEADER_ANALYSIS = "{{ config(materialized=\'table\', schema=\'observability_analysis\') }}\n"
TABLE_ANALYSIS = '{{get_table_analysis(source_table)}}'
COLUMN_ANALYSIS = '{{get_column_analysis(source_table)}}'
FIELD_ANALYSIS = '{{get_field_analysis(source_table)}}'
```

Ilustración 24 Declaración de variables

Hecho esto, iteraremos cada una de las tablas especificadas en la configuración, y crearemos un modelo por cada tipo de métrica, usando las variables que hemos declarado para simplificar el código lo máximo posible.

```
with  open(model_name+ '_tbl.sql', "w") as tbl_met:
    tbl_met.write(HEADER_METRIC)
    tbl_met.write(SOURCE_L.format(src1=model['object'][0], src2=model['object'][1]))
    tbl_met.write(TABLE_METRIC.format(metrics=model['metrics'], thresholds=model['thresholds']))

# Create column metrics
with  open(model_name+  '_col.sql', "w") as col_met:
    col_met.write(HEADER_METRIC)
    col_met.write(SOURCE_L.format(src1=model['object'][0], src2=model['object'][1]))
    col_met.write(COLUMN_METRIC.format(columns=model['columns']))

# Create field metrics
with  open(model_name+ '_fld.sql', "w") as fld_met:
    fld_met.write(HEADER_METRIC)
    fld_met.write(SOURCE_L.format(src1=model['object'][0], src2=model['object'][1]))
    fld_met.write(FIELD_METRIC.format(columns=model['columns'], meta_time=model['meta']['timestamp_column']))
```

Ilustración 25 Creación de modelos

Destacamos que tendremos que añadir un if para comprobar si la tabla fuente es un array (tabla de Snowflake) o un string (modelo de DBT), para ello usaremos la función **isinstance** para saber el tipo del parámetro, y haremos la llamada a la fuente en función de su tipo.

```
if isinstance(model['object'], list):
    model_name = OBSERVABILITY_PATH + model['object'][1] + suffix_obs
```

Ilustración 26 Caso tabla de origen

```
elif isinstance(model['object'], str):
    model_name = OBSERVABILITY_PATH + model['object'] + suffix_obs
```

Ilustración 27 Caso modelo de DBT

Finalmente, añadiremos una librería de Python llamada **Click**, que sirve para poder hacer ejecuciones de scripts vía línea de comandos, dándonos más opciones a la hora de ejecutar el script.

```
@click.command()
@click.option('--project', default='.', help='Path to the dbt project where observability models need to be added')
```

Ilustración 28 Parámetros de Click

Si el dbt_project estaba bien configurado, la ejecución será exitosa y se creará en la carpeta models del proyecto de DBT, una nueva subcarpeta llamada **observability_models**, donde estarán todos los modelos con las llamadas a las métricas, y que posteriormente ejecutaremos para crear las nuevas tablas.

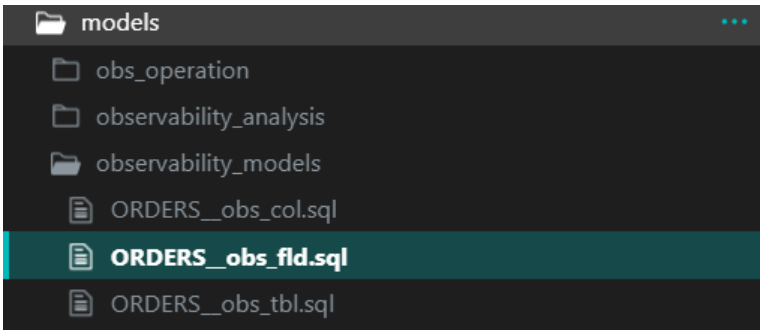

Ilustración 29 Carpeta de modelos creados

4.2.4. Modelos

Para estratificar el desarrollo y reducir al mínimo posible las llamadas a dicha variable, vamos a realizar una sola llamada por cada modelo que necesitemos, y que sean las macros que hemos creado las encargadas de iterar por dicha variable de forma automática. Haciéndolo de esta manera, no tendremos llamadas innecesarias, y se favorecerá la simplificación de las interacciones del usuario.

Siguiendo este sistema, necesitaremos tres modelos (uno por cada tipo de métrica) que llamen a las macros pertinentes por cada una de las tablas que queramos analizar. Estos modelos serán muy similares entre sí, y tendrán unas líneas de código donde declararemos opciones de configuración como el nombre del schema donde se guardará la tabla resultante, o si queremos que sea una tabla que se sobreescriba con cada ejecución o que sea incremental.

Aparte de estas líneas de configuración, los modelos usarán el lenguaje de markup **Jinja** para declarar como variable la tabla fuente, y luego pasará ese parámetro junto con el resto que necesite la macro.

```
{{ config(materialized='incremental', schema='observability_tables') }}
{%-set source_table = source("SAMPLE_OBSERVABILITY","ORDERS")-%}
{{get_field_metrics([{'column_set': None, 'columns': ['O_TOTALPRICE', 'O_CUSTKEY'], 'exclude_columns': [], 'metrics': [],
```

Ilustración 30 Modelo con llamada a métricas de campo

De cara al desarrollo de las macros, este proceso hace bastante más sencillo el iterar por las macros, ya que se les pasa el set de columnas completo, y la macro puede usar las funciones de Jinja para iterar como se necesite.

Cada uno de estos modelos generará una tabla incremental con el mismo nombre que el modelo, y que tendrá como contenido la tabla resultante al llamar a la macro con los parámetros suministrados.

4.2.5. Resultados

Una vez terminado el desarrollo de la primera capa, y ya creados todos los modelos y configurado el sistema, bastará con ejecutar en la consola de DBT el comando **dbt run –select**

models/observability_models, y todos los modelos de dicha carpeta crearán una tabla incremental o añadirán sus registros de la ejecución actual en caso de que ya existiera.

COLUMN_NAME	METRIC_NAME	METRIC_VALUE	UPPER_LIMIT	LOWER_LIMIT
O_TOTALPRICE	NULL_RATE	0	NULL	NULL
O_CUSTKEY	STANDART_DEV_POP	43304.474572842075	NULL	NULL
O_CUSTKEY	NULL_RATE	0	NULL	NULL
O_TOTALPRICE	STANDART_DEV_POP	88621.40182316916	NULL	NULL
O_CUSTKEY	MED	74990	NULL	NULL
O_CUSTKEY	MAD	37522	NULL	NULL
O_CUSTKEY	MINIMUM	1	1000	100

Ilustración 31 Tablas finales de primera capa

Estas tablas se generarán en Snowflake en el schema **dbt_rbenito_observability_tables**, y al ser incrementales y tener asociada la fecha de la ejecución, nos permitirán observar mediante consultas el histórico de datos que hemos tenido a lo largo del tiempo.

Con estos resultados, un usuario ya podría diseñar sus propias consultas y comprobar si ha sobrepasado los límites superiores o inferiores de forma manual, pero nos encargaremos en la segunda capa de que eso no sea una necesidad, sino una posibilidad, puesto que la segunda capa recogerá y mostrará todos los registros que han sobrepasado estos límites.

4.3. Segunda capa

Tal como mencionamos previamente, esta capa estará dedicada a analizar los registros generados en la primera, y aislar para poder identificar y tratar con mayor facilidad aquellos que no pasen los criterios que el usuario haya impuesto.

Es importante destacar que la mayoría de procesos que suceden en esta capa, son idénticos o muy similares a los que suceden en la primera, pero simplificados, por lo que nos centraremos principalmente las diferencias que hay entre ambas capas en lugar de indicar con detalle todo el proceso como hicimos en la anterior.

4.3.1. Dbt_project.yml

La configuración de esta capa también se da en el mismo archivo que la de la primera capa, aunque es considerablemente más breve y sencilla, ya que en este caso el usuario solo tiene que decidir qué límite superior o inferior aplicará a cada métrica, y sobre qué columnas quiere aplicar dichas restricciones.

```yaml
- column_set:
  columns: [O_TOTALPRICE,O_CUSTKEY]
  exclude_columns: []
  metrics: []
  exclude_metrics: ['AVERAGE','MOVING_AVERAGE']
  thresholds:
  - metric: 'MINIMUM'
    column_set:
    - columns: [O_TOTALPRICE,O_CUSTKEY]
      upper_limit: 1000
      lower_limit: 100
  - metric: 'MAXIMUM'
    column_set:
    - columns: [O_TOTALPRICE,O_CUSTKEY]
      upper_limit: 2000
      lower_limit: 200
```

Ilustración 32 Configuración de límites

Como podemos apreciar en la anterior ilustración, la configuración se declara a nivel de cada set de columnas, aplicándose exclusivamente a las que está configurando dicho set. Esto da libertad al usuario para poder ser tan general o específico como necesite con la configuración de las columnas, dándole la oportunidad de definir límites genéricos para unas, y otros más específicos para otras.

El siguiente nivel de granularidad de configuración viene dado por las métricas a las que se le aplican estos límites. Puede haber tantos bloques como métricas se vayan a analizar en dicho set de columnas, y posteriormente, encontramos otro bloque de sets de columnas, que permiten al usuario aplicar los límites solo a las columnas específicas que deseen en dicho set.

4.3.2. Macros

Al igual que en la anterior capa, necesitaremos en esta una macro por cada tipo de métrica que poseemos, las cuales tendrán como objetivo analizar el valor base de cada uno de los registros recogidos de las tablas de métricas y contrastarlos contra los límites especificados en el dbt_project.yml, para así guardar todos los que sobrepasen los límites en una tabla aparte.

Las tres macros son muy similares, pero tienen una diferencia que hace que prefiramos tener tres macros individuales en lugar de intentar hacer una genérica para el proceso. Comenzaremos por ver la de métricas de tabla.

```
{% macro snowflake__get_field_analysis(source_table) -%}
```

Ilustración 33 Cabecera de análisis de métricas de campo

Ya podemos apreciar viendo la macro la primera diferencia con las anteriores, y es que en este caso solo pasamos como parámetro la tabla objetivo a analizar. Esto es debido a que por cómo estructuramos las columnas de la anterior capa, ya encontramos los límites inferiores y superiores asociados a cada registro en caso de que los tengan, lo que implica que solo es necesario tomar esas dos columnas y compararlas contra la columna de **VALUE.**

```
(SELECT REFERENCE, COLUMN_NAME, COLUMN_VALUE, METRIC_NAME, METRIC_VALUE, 'UPPER_LIMIT' as SURPASSED_LIMIT, UPPER_LIMIT AS LIMIT_VALUE,
    (METRIC_VALUE-UPPER_LIMIT) as LIMIT_DIFF, current_timestamp() as TIMESTAMP
FROM last_reg
WHERE (UPPER_LIMIT IS NOT NULL) AND (METRIC_VALUE > UPPER_LIMIT))

UNION

(SELECT REFERENCE, COLUMN_NAME, COLUMN_VALUE, METRIC_NAME, METRIC_VALUE, 'LOWER_LIMIT' as SURPASSED_LIMIT, LOWER_LIMIT AS LIMIT_VALUE,
    (LOWER_LIMIT-METRIC_VALUE) as LIMIT_DIFF, current_timestamp() as TIMESTAMP
FROM last_reg
WHERE (LOWER_LIMIT IS NOT NULL) AND (METRIC_VALUE < LOWER_LIMIT))
```

Ilustración 34 Consulta de métricas de columna

La macro creará una query que busque todos los registros que tengan asociado un límite, y creará una tabla con todos aquellos que sobrepasen el límite mencionado, asociándoles como columnas la tabla de origen (**REFERENCE**), el nombre de la columna original, la métrica a analizar y el propio valor de la misma. Adicionalmente, también se indica el límite sobrepasado en cada registro y la cantidad por la que se ha sobrepasado.

Es muy importante destacar que esta macro solo queremos aplicarla a los últimos registros que se generaron en la primera capa, lo que necesitamos un bloque de código que haga que este tratamiento solo se aplique sobre aquellos registros generados en la última ejecución.

Para facilitar esta tarea, declararemos una tabla que contenga solo los registros más recientes mediante un **WITH** (función SQL usada para guardar una tabla en una misma query y referenciarla posteriormente). Para ello, nos valdremos de la forma que tiene DBT de aplicar los **timestamps**: Como DBT ejecuta todos los registros al mismo tiempo, todos los de la última ejecución de una tabla tendrán exactamente el mismo timestamp, por lo que solo tenemos que localizar el más reciente de dicha tabla, y filtrar todos los registros que sean menos recientes que esa fecha.

```
WITH last_reg as (
    (select *
    from {{source_table}}
    where timestamp = (select max(timestamp) from {{source_table}})))
```

Ilustración 35 Filtro de registros de la última ejecución

No será necesario entrar en más detalle con las otras dos macros, puesto que son idénticas a ésta, teniendo como única diferencia el nombre de las propias macros, y que en la de métricas de columnas añadimos como parámetro el nombre de la columna asociada al registro, y en la de métricas de campo añadimos el valor de cada registro individual.

4.3.3. Script de python

El script en esta capa es el mismo que en la anterior, y debe haberse ejecutado antes de hacer el run de la primera capa para así generar los modelos tanto de la primera como de la segunda.

La única diferencia que tiene con respecto a su comportamiento en la anterior capa, es que el sufijo que se añade a los modelos por defecto es **"__anly**" y que los modelos solo reciben como parámetro el nombre de la tabla objetivo de la primera capa.

```
# Obtain or set default suffix_anly
suffix_anly = observability.get('config',{}).get('analysis_table_suffix','__anly')
```

Ilustración 36 Variante del script para modelos de análisis

4.3.4. Modelos

Los modelos también serán muy similares a la anterior capa pero más simplificados. Como bien vimos en las macros, el único parámetro necesario para que estas funcionen será la tabla fuente, por lo que en este caso los modelos no necesitarán tener grandes porciones de código .yaml como sucedía en la capa previa.

```
{{ config(materialized='table', schema='observability_analysis') }}
{%-set source_table = ref("ORDERS__obs_col")-%}
{{get_column_analysis(source_table)}}
```

Ilustración 37 Modelo con macro de análisis de columnas

Si nos fijamos en la anterior ilustración, podemos apreciar una diferencia más en el parámetro de configuración **materialized**, ya que esta vez en lugar de usar como valor **'incremental'**, usa **'table'**.

La diferencia que esto produce en las tablas finales es que las que se generan en esta capa no son incrementales, y cada vez que ejecutemos esta capa, se sobreescribirán todas las tablas que había previamente con los resultados nuevos de esta ejecución. El motivo de esto es que realmente solo queremos usar esta capa para aislar los campos problemáticos de la última ejecución, y como esta capa está diseñada para ser ejecutada justo después de la primera, los registros antiguos no son necesarios.

En caso de querer revisar cómo de comunes eran los problemas de límites en registros anteriores, el usuario siempre tiene la posibilidad de hacer una query manual sobre los registros de la primera capa, ya que ahí sí que se conservan los registros previos, y en cada uno de ellos aparecen los límites, por lo que una query sencilla serviría para extraer un histórico de estos datos.

4.3.5. Resultados

Finalizada esta capa, solo quedaría comprobar el resultado. Lo primero sería ejecutar la misma función que usábamos para ejecutar la anterior capa, solo que ahora cambiaremos el parámetro que indicaba la carpeta a ejecutar para lanzar solo la asociada a los modelos de análisis: **dbt run –select models/observability_analysis.**

Destacamos que DBT tiene suficiente autonomía como para ordenar modelos a la hora de ejecutarlos y en caso de haber unos dependientes de otros, tiene la capacidad de lanzar primero los independientes y luego los dependientes, por lo que también podríamos lanzar las dos capas con un solo comando usando **dbt run –select models/observability_models models/observability_analysis.**

El resultado final de esta capa será una tabla por cada una que hubiese en la primera capa y que tuviese registros con límites definidos. Todos los registros que encontremos en estas tablas serán aquellos que han sobrepasado o bien el límite superior o inferior, y en la última capa, los usaremos para dar una representación visual de estos datos.

REFERENCE	COLUMN_NAME	METRIC_NAME	METRIC_VALUE	SURPASSED_LIMIT	LIMIT_VALUE	LIMIT_DIFF
SNOWFLAKE_SAMPLE_DATA.TP…	O_CUSTKEY	MAXIMUM	149999	UPPER_LIMIT	2000	147999
SNOWFLAKE_SAMPLE_DATA.TP…	O_TOTALPRICE	MAXIMUM	555285.16	UPPER_LIMIT	2000	553285.16
SNOWFLAKE_SAMPLE_DATA.TP…	O_CUSTKEY	MINIMUM	1	LOWER_LIMIT	100	99

Ilustración 38 Tablas finales segunda capa

4.4. Tercera capa

Esta será la capa final del framework, y será también la más distinta de las anteriores, ya que no intervendrá DBT en absoluto, Snowflake solo se usará al principio del proceso para asociar los registros de las tablas de las anteriores capas con la tercera, dándonos la posibilidad de recogerlos en **PowerBI** y crear un **dashboard** que nos ayude a mostrar de forma más visual y así extraer conclusiones de nuestros datos con mayor facilidad.

4.4.1. Preprocesado

La forma que tiene de funcionar PowerBI es primero tomar la información en bruto de las tablas de origen, realizarle unas transformaciones previas para adaptarlo al formato necesario para el dashboard, y posteriormente plasmar esa misma información de forma visual.

Este comportamiento es muy conveniente ya que los datos que se transforman son solo una copia de los datos de origen, por lo que se pueden realizar todos los ajustes necesarios para conveniencia de la aplicación visual, sin afectar a los datos originales.

Algunas de las transformaciones simples como cambiar el formato de una columna, o poner todos los textos de otra en minúsculas, se puede hacer mediante la interfaz visual, pero por lo general, las transformaciones más complejas se realizan mediante **DAX (Data Analytics eXpression).**

Los DAX son un conjunto de funciones que componen una especie de lenguaje de programación propio de PowerBI, destinado principalmente a realizar transformaciones de datos para alterar o generar tablas nuevas para nuestros dashboards.

En nuestro caso no necesitaremos crear demasiadas transformaciones, pero sí que tendremos que aplicar un DAX para crear una tabla nueva que sea una recolección de las tablas que vamos a analizar, y que nos muestre los últimos registros generados en la primera capa, y los generados en la segunda. Usando esta tabla, podemos extraer el porcentaje de los casos que han presentado problemas con respecto al total de casos totales.

El DAX hace un conteo de las filas de las tablas de la primera capa después de filtrar solo los últimos registros, y asocia a ese conteo otro extraído de su tabla de análisis asociada. Aparte de esto, añadimos también un carácter visual que representa un círculo, para luego poder asociarle un color en función del porcentaje de registros problemáticos.

```
TablaIndex = {(FIRSTNONBLANK('ORDERS__OBS_TBL (2)'[TableName],'ORDERS__OBS_TBL (2)'[TableName]),
            LASTNONBLANK('ORDERS__OBS_TBL (2)'[REFERENCE],'ORDERS__OBS_TBL (2)'[REFERENCE]),
            CALCULATE(COUNTROWS('ORDERS__OBS_TBL (2)'), FILTER('ORDERS__OBS_TBL (2)', 'ORDERS__OBS_TBL (2)'[TIMESTAMP]=MAX('ORDERS__OBS_TBL (2)'[TIMESTAMP]))),
            COUNTROWS('ORDERS__ANLY_TBL (2)'),
            UNICHAR(11044)),
            (FIRSTNONBLANK('ORDERS__OBS_COL (2)'[TableName],'ORDERS__OBS_COL (2)'[TableName]),
            LASTNONBLANK('ORDERS__OBS_COL (2)'[REFERENCE],'ORDERS__OBS_COL (2)'[REFERENCE]),
            CALCULATE(COUNTROWS('ORDERS__OBS_COL (2)'), FILTER('ORDERS__OBS_COL (2)', 'ORDERS__OBS_COL (2)'[TIMESTAMP]=MAX('ORDERS__OBS_COL (2)'[TIMESTAMP]))),
            COUNTROWS('ORDERS_ANLY_COL (2)'),
            UNICHAR(11044)),
            (FIRSTNONBLANK('ORDERS__OBS_FLD (2)'[TableName],'ORDERS__OBS_FLD (2)'[TableName]),
            LASTNONBLANK('ORDERS__OBS_FLD (2)'[REFERENCE],'ORDERS__OBS_FLD (2)'[REFERENCE]),
            CALCULATE(COUNTROWS('ORDERS__OBS_FLD (2)'), FILTER('ORDERS__OBS_FLD (2)', 'ORDERS__OBS_FLD (2)'[TIMESTAMP]=MAX('ORDERS__OBS_FLD (2)'[TIMESTAMP]))),
            COUNTROWS('ORDERS__ANLY_FLD (2)'),
            UNICHAR(11044))}
```

Ilustración 39 DAX para conteo de últimos registros

Hecho ya este preprocesado, podemos pasar directamente a la parte del dashboard, el cual está dividido en **tres niveles de profundidad**: Uno general a modo de vista de pájaro de todas las tablas junto con el total de registros erróneos de cada una; otro más específico para cada tabla asociado a cada una de sus tres tipos de métricas posibles; y el de más bajo nivel, el cual toma todas toda la información del histórico de una métrica asociada a una tabla y la compara contra los registros erróneos de la última ejecución.

4.4.2. Primer nivel

El primer nivel mostrará la tabla que hemos creado previamente usando un DAX, recogiendo en cada fila cada tabla individual generada en la primera capa, y mostrando la tabla de origen, los registros totales y los registros erróneos.

Además, añadiremos un par de campos nuevos: Uno para indicar el **Promedio de Error_rate**, expresando en un número entre 0 y 1 el porcentaje de casos del total que han tenido errores.

El otro campo será el **Check,** asociado al carácter circular que añadimos previamente, y al cual, mediante funciones de PowerBI, asignaremos un color dependiendo de como de alto sea el Error_rate, siendo verde en caso de que sea 0, naranja si es mayor que 0 pero menor que 0.20, y rojo si es mayor o igual a 0.20.

Aparte de esta tabla, también mostramos un gráfico de barras representando el Error_rate de cada una de las tablas, para ayudarnos a contrastar cómo de común es que tengamos un Error_rate alto en una tabla con respecto a las demás.

Debajo de ese mismo gráfico poseemos también varios botones, los cuales nos llevan hacia las siguientes pestañas de la aplicación, las cuales se corresponden con los siguientes niveles de profundidad.

Por desgracia, por limitaciones de PowerBI, no podemos asociar estos botones a cada uno de los elementos del gráfico de barras, ni tampoco podemos asociar estos enlaces a registros individuales dentro de la tabla, por lo que tienen que ser botones estáticos que colocamos manualmente para ayudarnos a navegar por la aplicación.

Últimos registros

Table_name	Reference	Registros_totales	Registros_erroneos	Promedio de Error_rate	Check
ORDERS__OBS_COL	SNOWFLAKE_SAMPLE_DATA.TPCH_SF1.ORDERS	18	3	0,17	
ORDERS__OBS_FLD	SNOWFLAKE_SAMPLE_DATA.TPCH_SF1.ORDERS	8223626	1500000	0,18	
ORDERS__OBS_TBL	SNOWFLAKE_SAMPLE_DATA.TPCH_SF1.ORDERS	3	1	0,33	
Total				**0,23**	

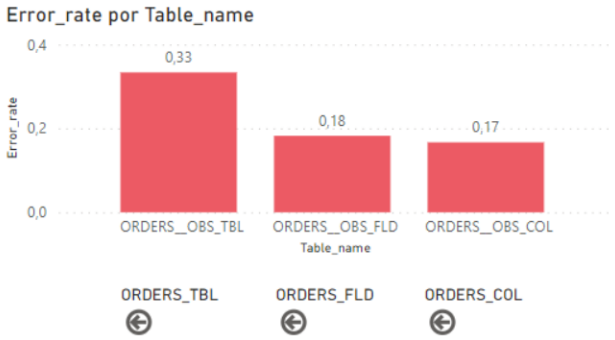

Ilustración 40 Visuales del primer nivel

4.4.3. Segundo nivel

El siguiente nivel de granularidad estudia cada una de las tablas que veíamos en el anterior nivel con mayor detalle, permitiéndonos ver los registros de la última ejecución tanto de las tablas de métricas junto a las de análisis, dándonos así la posibilidad de localizar con mayor facilidad qué registros son los que están sobrepasando los límites.

En este caso, se plasman las mismas columnas que tenemos en Snowflake en la primera y en la segunda capa, y añadimos también un gráfico de barras para mostrar el total de registros de la última ejecución contra el total de registros fallidos.

Nótese que habrá una pestaña de este nivel por cada tabla de métricas que haya en Snowflake, por lo que en el caso de esta demostración, nosotros contaremos con tres pestañas.

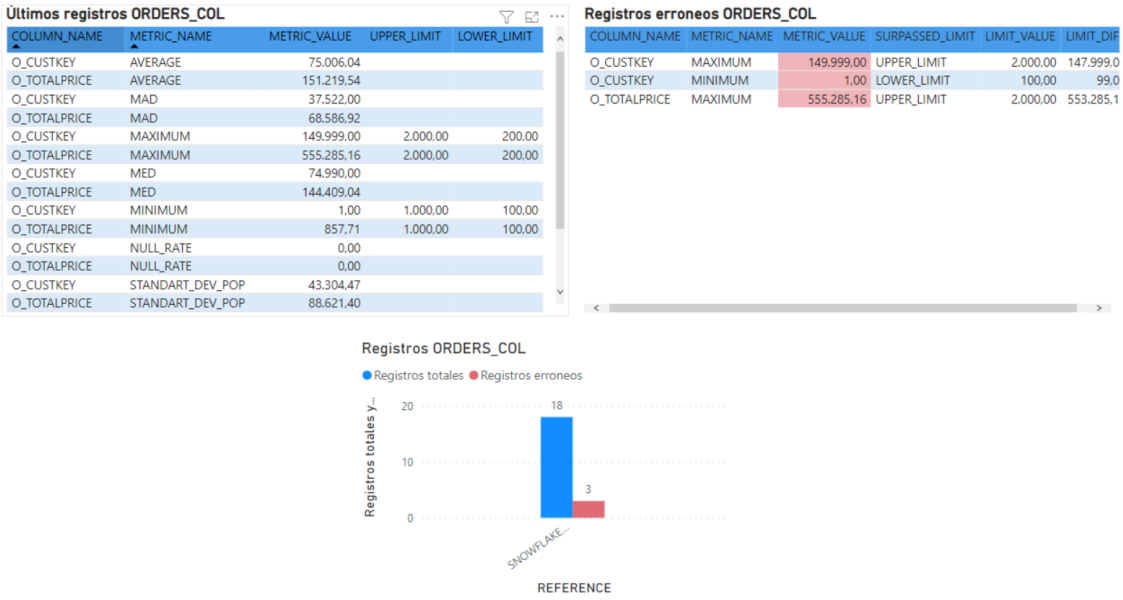

Ilustración 41 Visuales del segundo nivel

4.4.4. Tercer nivel

Este será el nivel más bajo de profundidad que encontraremos en PowerBI, no solo podremos ver en él los casos totales contra los casos erróneos, sino que también podremos ver el histórico de registros asociados a la métrica que estamos analizando con respecto a una tabla.

Dispondremos de un gráfico de líneas, donde podremos ver los límites superiores e inferiores a lo largo del tiempo, junto con la métrica en cuestión, de modo que si la métrica se cruza en algún punto del gráfico con una de estas dos líneas, sabremos que en ese momento hubo un rebasamiento del límite.

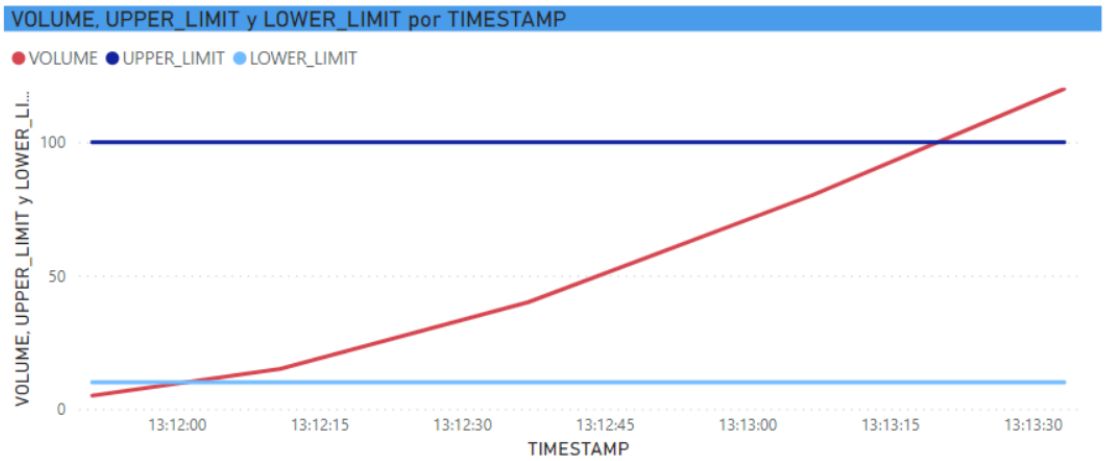

Ilustración 42 Gráfico de líneas del tercer nivel

Además de este gráfico, contamos con dos tablas que nos darán algo más de información sobre el histórico. La primera de ellas recogerá todos los registros que hay en la base de datos de esa métrica, y mostrará su valor, junto con sus límites superiores e inferiores, marcando en rojo aquellos límites que fueron sobrepasados en algún momento.

Total registros VOLUME

REFERENCE	METRIC	VALUE	UPPER_LIMIT	LOWER_LIMIT	TIMESTAMP
SNOWFLAKE_SAMPLE_DATA.TPCH_SF1.ORDERS	VOLUME	5,00	100,00	10,00	11/04/2022 13:11:51
SNOWFLAKE_SAMPLE_DATA.TPCH_SF1.ORDERS	VOLUME	15,00	100,00	10,00	11/04/2022 13:12:10
SNOWFLAKE_SAMPLE_DATA.TPCH_SF1.ORDERS	VOLUME	40,00	100,00	10,00	11/04/2022 13:12:37
SNOWFLAKE_SAMPLE_DATA.TPCH_SF1.ORDERS	VOLUME	80,00	100,00	10,00	11/04/2022 13:13:06
SNOWFLAKE_SAMPLE_DATA.TPCH_SF1.ORDERS	VOLUME	120,00	100,00	10,00	11/04/2022 13:13:33
Total					

Ilustración 43 Histórico del VOLUME

La última tabla es la más simple, mostrando únicamente los registros erróneos, con los mismos campos que encontramos en las tablas de análisis de Snowflake, dándonos la posibilidad de localizar estos casos con facilidad en caso de que tengamos un histórico muy amplio.

REFERENCE	METRIC	VALUE	SURPASSED_LIMIT	LIMIT_VALUE	LIMIT_DIFF	TIMESTAMP
SNOWFLAKE_SAMPLE_DATA.TPCH_SF1.ORDERS	VOLUME	120,00	UPPER_LIMIT	100,00	20,00	11/04/2022 14:07:50

Ilustración 44 Registros erroneos en última ejecución

5. SEGUIMIENTO Y CONTROL

Ya terminado el proyecto, vamos ahora a hacer un análisis de la planificación inicial y compararla contra los acontecimientos sucedidos a lo largo del proyecto.

5.1. Alcance

Empezamos por analizar la tabla de alcance junto con cada uno de los requisitos que propusimos inicialmente:

Código	Descripción	
RE-01	Se deben poder manejar grandes cantidades de información de forma dinámica.	✓
RE-02	El usuario debe poder configurar la aplicación según sus necesidades.	✓
RE-03	Se deben poder tratar registros de todos los tipos de datos aceptados en Snowflake	✓
RE-04	La aplicación debe admitir cambios en la configuración entre ejecuciones.	✓
RE-05	Los registros de la base de datos deben representarse automáticamente en PowerBI sin necesidad de interacción por parte del usuario.	✓
RE-06	El proceso debe estar lo bastante automatizado para que el usuario no necesite más conocimientos que los requeridos para configurar la ejecución.	✓
RE-07	La configuración de las ejecuciones debe estar contenida en un solo archivo dentro de DBT.	✓

RE-08	Debe haber métricas asociadas a los pilares de Freshness, Schema, Volume y Distribution.	

Todos los requisitos que propusimos se cumplen satisfactoriamente, aunque hacemos un inciso en dos de estos requisitos:

- RE-01: Si bien podemos manejar grandes cantidades de registros y tablas, la parte relacionada con la tercera capa (la visual) es bastante estática, ya que requiere que el usuario diseñe una diapositiva manualmente por cada tabla y métrica que quiere observar. Aunque este proceso es bastante rápido ya que se puede copiar y pegar de las anteriores pestañas, no está pensado para tener una pestaña por cada métrica en caso de estar analizando muchas tablas con muchas métricas aplicadas a cada una de ellas.
- RE-03: Todos los tipos de datos de Snowflake pueden ser procesados, pero solo los numéricos pueden utilizar todas las métricas disponibles. Los no numéricos pueden utilizar algunas métricas, pero solo un número reducido de estas.

Aun teniendo en cuenta estos incisos, seguimos considerando que estos dos requisitos han sido cumplimentados, por lo que consideraremos esta sección como un éxito.

5.2. Cronograma

Veremos primero el cronograma inicial que teníamos al principio del proyecto:

Hitos	S5	S6	S7	S8	S9	S10	S11	S12	S13	S14	S15	S16	S17	S18	S19	S20
T-1 Análisis																
T-2 Diseño																
T-3 Implementación																
T-4 Interfaz gráfica																
T-5 Memoria																

Ahora vamos a compararlo contra el cronograma del tiempo real que hemos dedicado a cada sección:

Hitos	S5	S6	S7	S8	S9	S10	S11	S12	S13	S14	S15	S16	S17	S18	S19	S20	S21
T-1 Análisis																	
T-2 Diseño																	
T-3 Implementación																	
T-4 Interfaz gráfica																	
T-5 Memoria																	

Como podemos observar, ha habido algunas desviaciones, principalmente en las fases de análisis y diseño, para las cuales tuvimos que dedicar una semana extra para cada una de ellas. La causa del retraso en el análisis es que el planteamiento del proyecto propuesto en la primera semana acabó por ser descartado por completo en pos de orientar el TFG en un área no relacionada con el mundo Open Source.

Con respecto al diseño, el framework que estábamos planteando entre Snowflake, DBT y PowerBI, resultó ser más complicado de lo que esperábamos en un principio, por lo que necesitamos una semana extra para identificar algunos puntos críticos que íbamos a encontrar posteriormente, y a plantear como íbamos a enfrentarlos.

Finalmente, también acabamos por dedicar una semana más al desarrollo, ya que algunas de las propuestas del diseño acabaron por cambiar, resultando en modificaciones en el código ya desarrollado, lo cual, tras suceder un par de veces, acabó por provocar una semana de retraso, aunque en este caso es menos acuciante que en los anteriores, porque la desviación es menor al tener planteado originalmente unos tiempos estimados mayores que en las anteriores secciones.

No obstante, sí que encontramos algo positivo en las etapas de la Interfaz gráfica y Memoria, ya que nos llevó una semana menos de lo esperado realizar cada una de ellas, la interfaz gráfica porque PowerBI es muy amigable con usuarios novatos y tiene una interfaz muy cómoda, y la memoria porque la redacción de la misma fluyó bastante rápido considerando las horas dedicadas a la misma, especialmente a partir de la segunda mitad.

El resultado final es que ha sido necesaria una semana extra para terminar el proyecto al completo, lo cual equivale a una desviación del 5% del tiempo total estimado, y dado que era el primer proyecto de esta escala que estaba realizando y la complejidad del mismo, estamos satisfechos con el resultado.

5.3. Gestión de riesgos

Mencionaremos ahora aquellos riesgos que intentamos predecir al principio del proyecto y que acabamos por encontrarnos, dejando de lado aquellos con los que no nos hemos topado:

- **RI-3 Decisiones erróneas de diseño:** La principal es que en un principio planteamos distribuir las métricas de las tablas por columnas, pero el debate de si debería ser por filas o por columnas se repitió múltiples veces, haciendo que oscilasemos esta decisión varias veces a lo largo del proyecto, decidiendo finalmente dejarlo en distribución por filas.
- **RI-6 Mala gestión del tiempo:** Este riesgo se encontró principalmente en la etapa de Diseño, ya que el planteamiento inicial del framework fue más complicado de lo que esperábamos, y acabó por llevar una semana más de lo esperado.

Aparte de estos dos riesgos, encontramos un problema más en el desarrollo del proyecto que pensamos previamente que nos íbamos a encontrar:

- **RI-9 Caducidad de la cuenta de Snowflake:** Al estar usando una versión de prueba de Snowflake, esta llegó un punto donde caducó, haciendo que tuviésemos que migrar los datos de una cuenta a otra, lo cual resultó ser más complicado de lo esperado debido a los problemas que da Snowflake al ingestar tablas que tienen archivos .json en algunas celdas.

Ya mencionados estos riesgos y teniendo en cuenta la sección anterior, podemos ver que no ha habido demasiadas desviaciones, por lo que, a pesar de que surgió un problema que no estaba recogido en la gestión de riesgos, en general ha sido bastante acertada.

6. CONCLUSIONES FINALES

6.1. Estado final del proyecto

El framework que hemos diseñado tiene la capacidad de tomar registros de cualquier tabla y extraer rápidamente métricas y análisis de la misma, permitiendo al usuario monitorizar la calidad de sus datos requiriendo solo un mínimo esfuerzo en el fichero de configuración del framework, y permitiendo al usuario personalizar el análisis tanto como necesite.

Con el proyecto ya terminado, puedo decir que tanto el cliente como yo estamos muy satisfechos con el resultado final. Todos los requisitos han sido cumplimentados satisfactoriamente y la desviación del tiempo invertido en el mismo ha sido bastante reducida.

El trabajo ha sido desafiante, ya que hemos estado usando herramientas con las que teníamos poca o ninguna experiencia al inicio del proyecto, pero hemos sido capaces de hacernos con ellas y de manejarlas al nivel apropiado para poder alcanzar todas las metas que nos propusimos.

No obstante, esta es solo una versión inicial del proyecto, puesto que aún hay mucho margen de mejora, ya sea en la cantidad de métricas que estamos extrayendo, pudiendo añadir algunas más orientadas a campos no numéricos; o bien en la parte de análisis, donde podríamos tratar de crear algunos procesos de mayor complejidad para una detección más dinámicas de los casos problemáticos.

En ese sentido, el framework queda en manos del cliente, de modo que podrá llevarlo a otros proyectos propios y adaptarlo y extenderlo tanto como necesite.

6.2. Conocimientos adquiridos

De este proyecto hemos adquirido muchos conocimientos nuevos, algunos técnicos, y otros relacionados con la gestión del proyecto.

Con respecto a los conocimientos técnicos, hemos aprendido a manejar con algo más de profundidad Snowflake, y también PowerBI de forma algo más superficial; pero donde más conocimientos hemos adquirido, ha sido con DBT, donde hemos tenido que centrar la gran mayoría de nuestros esfuerzos de desarrollo, lo cual nos ha permitido adquirir conocimientos extensos de la herramienta, especialmente con respecto a la gestión de las macros, parte fundamental para la creación de paquetes en la aplicación.

Además de esto, también hemos ampliado considerablemente nuestros conocimientos de SQL, y en menor medida de Python y YAML, ya que hemos tenido que estar usando estos lenguajes continuamente a lo largo del proyecto, y ha sido necesario realizar algunas consultas de un nivel bastante superior a lo que estábamos acostumbrados previamente.

Finalmente, pasamos a los conocimientos relacionados con la gestión de proyectos. Puesto que es el primer proyecto de estas dimensiones que hemos efectuado, ha sido bastante distinto a los proyectos a los que estábamos acostumbrados en la etapa estudiantil. Al ser un proyecto tan grande, nos ha hecho darnos cuenta de la importancia que poseen las etapas de análisis y diseño en la realización de un proyecto grande, puesto que si las hubiésemos ignorado y hubiésemos pasado directamente a la parte de desarrollo, muy probablemente habríamos tardado incluso más en acabar el proyecto, y habríamos tenido que excluir algunos requisitos del mismo por la falta de tiempo.

6.3. Comentarios finales

Pasando a una parte más personal, me gustaría decir que siento que este proyecto ha marcado un antes y un después en mi vida como ingeniero informático, puesto que ha sido el punto de inflexión para poner a prueba todo lo aprendido a lo largo de la carrera.

Aunque ha habido algunos momentos de incertidumbre o de vértigo al tener una tarea tan grande en comparación con lo que estábamos acostumbrados previamente, ha sido muy satisfactorio el ver cómo cada pequeño desafío que encontrábamos ha acabado por ser superado, hasta alcanzar la versión final del proyecto.

Por último, me gustaría realizar unos agradecimientos especiales a todas aquellas personas que me han ayudado con el proyecto, empezando por supuesto por mis dos tutores: César Domínguez, mi tutor académico, el cual siempre ha estado dispuesto a recibirme y orientarme en la dirección correcta; y Carlos Acedo, mi tutor de la empresa, quien no solo me ha ayudado con el proyecto, sino que me ha ayudado a sentirme como uno más en la empresa.

También quiero dedicar agradecimientos especiales a todos los miembros del equipo de SDG que me han aconsejado y ayudado a llevar el proyecto adelante: Aarón Otero, Ángel González, Carlos Suárez, Itaitz Montalbán, Rubén Caparros y Xabier Gil.

BIBLIOGRAFÍA

Artículos:

1. **¿Qué es Data Observability?** https://turingears.com/que-es-data-observability/
2. **¿Pilares de Data Observability?** https://towardsdatascience.com/introducing-the-five-pillars-of-data-observability-e73734b263d5
3. **Data Observability con SQL 1** https://levelup.gitconnected.com/data-observability-building-your-own-data-quality-monitors-using-sql-a4c848b6882d
4. **Data Observability con SQL 2** https://levelup.gitconnected.com/data-observability-in-practice-using-sql-part-ii-schema-lineage-3ce07c759f18
5. **Cálculos estadísticos con SQL 1** https://sql-snippets.count.co/t/outlier-detection-in-snowflake/194
6. **Cálculos estadísticos con SQL 2** https://towardsdatascience.com/statistical-techniques-for-anomaly-detection-6ac89e32d17a
7. **DBT profiler** https://github.com/data-mie/dbt-profiler#print_profile_schema-source
8. **Diseñar paquetes de DBT** https://discourse.getdbt.com/t/an-approach-to-building-dbt-packages/1445
9. **Media sobre tiempo** https://learnsql.com/blog/moving-average-in-sql/

Documentación:

1. **Re_data** https://docs.getre.io/latest/docs/introduction/whatis
2. **DBT** https://docs.getdbt.com/docs/building-a-dbt-project/documentation
3. **Snowflake** https://docs.snowflake.com/en/
4. **PowerBI** https://docs.microsoft.com/es-es/power-bi/

5. **DAX** https://docs.microsoft.com/es-es/power-bi/transform-model/desktop-quickstart-learn-dax-basics