TRABAJO FIN DE GRADO

Grado en Ingeniería Informática

UNIVERSIDAD DE ALMERIA

ESCUELA SUPERIOR DE INGENIERÍA

Análisis del tráfico rodado en Bélgica usando Azure

Curso 2019/2020

Alumno/a:

José Manuel Martínez Salas

Director/es:

José Joaquín Cañadas Martínez

José Pablo Cabeza García

IN LUMINE UNIVERSITAS

Agradecimientos

Ha sido un largo camino hasta llegar a este momento. El final de un tramo para continuar el camino. Pero durante este tramo han sido varias las personas que se han cruzado en mi camino para ayudarme a conseguir esta primera meta en mi camino.

En primer lugar, quiero agradecer a mi familia el apoyo y ayuda que me han ofrecido durante todo este camino en todo momento. A mi madre Charo por haber estado siempre confiando en mí y apoyándome en todas las decisiones que tomaba, las compartiera o no. A mi padre Manuel por haberme aconsejado y dado calma en momentos donde no sabía qué hacer ni qué decisión tomar. Y en tercer lugar a mi hermano menor, Dani, que, aunque estando en una edad muy mala, él siempre ha querido distraerme cuando lo he necesitado (y cuando no) para que jugase con él o para contarme historias que le habían ocurrido estando siempre ahí para sacar una carcajada cuando menos te lo esperas.

En segundo lugar, quiero también agradecer a mi pareja, por haber estado al lado mío durante todo este camino, por aguantarme durante mis peores momentos de estrés y agobio donde a pesar de no demostrar mucho esos sentimientos ella sabía perfectamente cómo estaba yo. Por escucharme y darme consejo cuando lo necesitaba. Un pilar fundamental y una fuente de motivación especial.

En tercer lugar, me gustaría agradecer a mis compañeros de carrera Juanjo Escarabajal, Juan Merlos, Antonio Jesús Gómez, Rafael Alejandro, Manuel Marín, Daniel Barroso, Guillermo y Miguel por todos estos años de experiencias, donde hemos compartido risas, quejas, sufrimientos y más risas aún.

En cuarto lugar, quiero agradecer a Pablo Cabeza, compañero de trabajo desde hace unos meses, por haberme enseñado con tranquilidad y tiempo muchos de los conceptos que he podido aplicar en este proyecto. Gracias a él y a su forma de enseñarme conceptos que eran nuevos para mí por no estar medito en el mundo del Big Data y la nube, he podido no solo aprender cómo se usan muchas tecnologías, sino comprender para que se usan y los beneficios de utilizarlas en cada situación para conseguir exprimir al máximo cada herramienta.

En quinto lugar, quiero agradecer a mi director de proyecto, José Joaquín Cañadas y a Manel Mena, por la forma en que nos introdujeron la nube, no como objetivo principal, sino como herramienta de apoyo a otros conceptos. Ellos fueron los que me hicieron empezar en este mundo y captaron mi atención en el mundo de la nube.

Y, por último, agradecer a Elastacloud, por haberme dado la oportunidad de entrar al mundo laboral en este ámbito y haberme acogido de tal forma, enseñándome y poniéndome en proyectos reales a trabajar desde el primer momento, demostrando confianza en mí. Elastacloud me ha mostrado los múltiples usos de la nube, en muchos ámbitos diferentes y me ha adentrado en una comunidad donde puedes encontrar a gente muy interesante.

ÍNDICE DE CONTENIDOS

Resumen
Abstract
1. Introducción 1
1.1 Motivación 1
1.2 Contextualización del problema a resolver 2
1.3 Objetivos 3
1.4 Planificación del proyecto 4
1.5 Estructura del documento 6
2. Estado del arte 8
2.1 Waze 8
2.2 Google Maps 9
2.3 Sistemas de la Dirección General de Tráfico 10
2.4 Mediatel. Route API 10
2.5 Conclusiones 11
3. Entendiendo el entorno del problema 12
3.1 Analizando el problema 12
3.2 Fuentes de datos 13
3.2.1 Listado de sensores 14
3.2.2 Información de los sensores 15
3.3 Planificando la solución 16
4. Tecnologías, herramientas y servicios 21
4.1 Tecnologías 21
4.1.1 Python 21
4.1.2 Transact-SQL 22
4.2 Herramientas 22
4.2.1 Visual Studio Code 22
4.2.2 Azure 23
4.2.3 Git 24
4.2.4 Azure Storage Explorer 24
4.2.5 Terraform 25
4.2.6 Spark 26
4.2.7 Structured Streaming 30
4.2.8 Power BI 32
4.2.9 Delta Lake 33
4.3 Servicios 37
4.3.1 Azure DevOps 37
4.3.2 Azure Data Factory 38
4.3.3 Azure SQL Database 39
4.3.4 Azure Databricks 39
4.3.5 Azure Function App 40
4.3.6 Azure Key Vault 41
4.3.7 Azure Stream Analytics 42
4.3.8 Azure Storage Account 43

4.3.9 Azure Event Grid 44
5. Arquitectura Delta 46
5.1 ¿Qué es la arquitectura Delta? 46
5.2 Otras arquitecturas para soluciones basada en flujos de datos 47
5.3 Ventajas de la arquitectura Delta 49
6. Traffic Monitoring. La plataforma de datos 53
6.1 Introducción, ¿qué es Traffic Monitoring? 53
6.2 Modelo de datos. Esquema en estrella 54
6.2.1 ¿Qué es? 54
6.2.2 Tablas de dimensiones 55
6.2.3 Tablas de hechos 55
6.3 Ingestión de los datos 56
6.3.1 Entradas de datos 56
6.3.2 Obteniendo los datos 57
6.3.3 Almacenando los datos obtenidos 60
6.4 Procesamiento de los datos en Databricks 60
6.4.1 Procesando la capa de Landing 60
6.4.2 Preparando la capa de Staging 63
6.4.3 Capa de Agregación: Estado del tráfico 66
6.4.4 Capa de Agregación: Estadísticas de afluencia de gente 67
6.5 Procesamiento de los datos en Streaming 70
6.5.1 Conjuntos de datos de entrada 70
6.5.2 Conjuntos de datos de salida 71
6.5.4 Query 71
6.6 Automatizando el proceso de despliegue 73
6.6.1 Infraestructura como código. Terraform 73
6.6.2 DevOps: Pipelines para construir la solución 75
6.6.3 DevOps: Pipelines para desplegar la solución 76
7. Resultados. Visualizando los datos procesados en Power BI 78
7.1 Informe en tiempo real del estado de los sensores 78
7.2 Informe en tiempo real del estado del tráfico 82
7.3 Informe de afluencia de personas para vender espacio publicitario 83
8. Conclusiones y trabajo futuro 84
8.1 Conclusiones 84
8.2 Trabajo futuro 85
Bibliografía 86

ÍNDICE DE FIGURAS

Figura 1. Tareas completadas en DevOps. 4
Figura 2. Planificación final de sprints. 5
Figura 3. Logotipo de Waze. 8
Figura 4. Selección de vista modo Tráfico. 9
Figura 5. Imagen de sensor DGT. 10
Figura 6. Logotipo de Route. 10
Figura 7. Ejemplo de respuesta API. Descripción del sensor. 14
Figura 8. Ejemplo de respuesta API. Mediciones del sensor. 15
Figura 9. Logotipo de Microsoft Azure. 16
Figura 10. Esquema de arquitectura a construir. 19
Figura 11. Diagrama de Arquitectura Delta. 20
Figura 12. Logotipo de Python. 21
Figura 13. Logotipo de T-SQL. 22
Figura 14. Interfaz y logotipo de VS Code. 23
Figura 15. Logotipo de Git. 24
Figura 16. Interfaz Azure Storage Explorer. 25
Figura 17. Logotipo de Terraform. 26
Figura 18. Flujo de trabajo en Spark. 27
Figura 19. Interacción entre componentes en Spark. 28
Figura 20. Estructura de un dataframe. 28
Figura 21. Transformación simple en Spark. 29
Figura 22. Transformación compleja en Spark.. 29
Figura 23. Structured Streaming ingestion. 31
Figura 24. Diagrama de trabajo. Structured Streaming. 32
Figura 25. Logotipo de Power BI. 33
Figura 26. Logotipo de Delta lake. 33
Figura 27. Diagrama sobre como Parquet almacena los datos. 34
Figura 28. Log de transacciones de un Delta 35
Figura 29. Delta Table. Cambios contenidos en archivos del log de transacciones. 35
Figura 30. Ejemplo de tabla de versiones con los últimos commits realizados. 36
Figura 31. Características clave de Delta Lake. 37
Figura 32. Logotipo de Azure DevOps. 38
Figura 33. Logotipo de Databricks. 40
Figura 34. Logotipo Azure Functions. 41
Figura 35. Logotipo Azure Key Vault. 42
Figura 36. Flujo de trabajo de Steam Analytics. 43
Figura 37. Servicios dentro de Azure Storage Account. 44
Figura 38. Flujo de trabajo con Event Grid. 45
Figura 39. Arquitectura Lambda. 48
Figura 40. Arquitectura Kappa. 49
Figura 41. Comparación entre arquitectura Lambda o Kappa vs Delta. 50
Figura 42. Ventajas de usar la arquitectura Delta. 52

Figura 43. Esquema en estrella. 54
Figura 44. Modelo de datos 56
Figura 45. Azure Function. Input Binding. 57
Figura 46. Obteniendo las variables de entorno. 58
Figura 47. Configuración de la Azure Function. 58
Figura 48. Log de la ingestión de datos. 60
Figura 49. Leer Stream con Structured Streaming utilizando abs-aqs. 61
Figura 50. Escribiendo stream con Structured Streaming. 61
Figura 51. Función merge para escribir un stream. 63
Figura 52. Esquema tabla de Landing. 64
Figura 53. Inferir esquema de datos automáticamente. 64
Figura 54. Lógica para calcular el estado del tráfico. 67
Figura 55. Lógica para calcular el número de personas por punto y hora. 69
Figura 56. Filtrando directorios para obtener archivos en Stream Analytics. 70
Figura 57. Conjuntos de datos de entrada y salida en Stream Analytics. 72
Figura 58. Deserializando elementos en Stream Analytics. 72
Figura 59. Estructura de archivos de Terraform 74
Figura 60. Ejecución del *pipeline* para construir la solución. 75
Figura 61. Creando un Service Principal. 76
Figura 62. Pipeline de despliegue en DevOps. 77
Figura 63. Informe estado de los sensores. Vista general. 78
Figura 64. Filtrado de datos en informe de estado de los sensores. 79
Figura 65. Recuento de sensores minuto a minuto. 80
Figura 66. Recuento de sensores que están enviando datos nuevos. 80
Figura 67. Recuento de sensores que están disponibles. 81
Figura 68. Recuento de sensores agrupados por regularidad de los datos. 81
Figura 69. Informe sobre estado del tráfico. 82
Figura 70. Informe de afluencia de personas por carretera y hora. 83

Resumen

Hoy en día el Big Data y la nube están en la mayoría de los servicios que utilizamos sin que ni siquiera seamos conscientes. Son una necesidad para el continuo desarrollo a día de hoy.

Traffic Monitoring se trata de una plataforma de datos con la cual se procesan los datos que provienen de sensores de tráfico rodado que el gobierno belga tiene instalados a lo largo de sus carreteras.

El sistema es capaz de procesar millones de registros por hora a un coste muy bajo a la vez que es capaz de servir resultados e información relevante en un periodo de tiempo muy bajo manteniendo una latencia aceptable a la hora de ofrecer la información en tiempo real.

El producto ha sido desarrollado utilizando tecnologías que están siendo utilizadas por las compañías más punteras del mundo en el momento de su desarrollo como son BP, Alibaba, Tencent Games, McAfee o UpWork como pueden ser Delta Lake o Spark.

Abstract

Nowadays the Big Data and Cloud are in most of the services we use, and we aren’t aware of it. They are a necessity for our continuous development in the world today.

Traffic Monitoring is a data platform built in Azure that processes data retrieved from traffic sensors installed in the Belgium roads by the government.

The system is capable of processing millions of records per hour at a very low cost while at the same time it is able to serve relevant results and information in a very low period of time while maintaining an acceptable latency in providing the information in real time.

The product has been developed using technologies that are being used by the world’s leading companies such as BP, Alibaba, Tencent Games , McAfee or UpWork Some of the technologies are Delta Lake and Spark.

1. Introducción

El presente Trabajo de Fin de Grado (TFG) va a consistir en el desarrollo de una plataforma de datos haciendo uso de los servicios en la Nube que proporciona Microsoft Azure para poder solucionar un problema propuesto.

1.1 Motivación

Los conceptos de la "Nube" y el "Big Data" están cada vez más presentes en nuestro día a día. Con el transcurso de los años más y más empresas comienzan la transición de su negocio hacia la Nube por las ventajas que esta ofrece.

Pero la Nube realmente existe desde hace bastantes años aunque no se tenía constancia de ellos. En los años 90 algunas empresas de telecomunicaciones empezaron a ofrecer las primeras VPNs como servicio cloud. Microsoft por ejemplo, con sus primeros sistemas operativos puso a disposición su servicio cloud de "Windows Update" con el cual los usuarios que hacían uso de su sistema operativo podían actualizar su ordenador sin necesidad de tener que emplear un disquete o CD.

A día de hoy, estos servicios están mucho más integrados en nuestra sociedad obviamente, pero no nos damos cuenta de que vivimos en un mundo donde prácticamente todo es parte de la nube.

Podríamos decir que un servicio cloud o servicio en la nube es un servicio que es entregado bajo demanda a una compañía o a una persona en particular a través de internet. Habitualmente estos servicios están diseñados para ser fácilmente accesibles por aplicaciones sin la necesidad de tener que proporcionar la infraestructura.

Al no existir la necesidad de tener que montar una infraestructura, aunque sea sencilla, la barrera de entrada que supone construir cualquier servicio es mucho menor, permitiendo a muchas personas y entidades acceder al mundo empresarial y difundir su servicio.

Además, ahora mismo la información es poder. Las empresas almacenan cantidades ingentes de datos cada día. Pero estos datos en sí no tienen ningún valor. Es necesario realizar un procesamiento y análisis de estos datos para poder extraer información relevante para el negocio. Esto es el llamado Big Data, la capacidad para construir arquitecturas de datos que sean capaz de procesar grandes cantidades de datos para ser analizadas y poder ayudar en la toma de decisiones del negocio.

Es aquí donde puedo extraer la motivación principal de este TFG. Quiero construir una plataforma de datos haciendo uso de la nube (en este caso Microsoft Azure) para procesar grandes cantidades de datos y con ello poder mostrar en un reporte información que ayude a una posible toma de decisiones o monitorización de un sistema. Con esto además, quiero mostrar también cómo podría ser un flujo de trabajo en la nube y algunos ejemplos de arquitecturas de software para procesamiento de datos.

1.2 Contextualización del problema a resolver

Para realizar este proyecto vamos a imaginarnos el siguiente problema. El gobierno de Bélgica tiene instalado en sus carreteras una serie de sensores de tráfico con los cuales toma medidas minuto a minuto. Esta información es enviada a sus propios servidores y luego es expuesta de manera pública a través de una API. Podemos descargar esta información minuto a minuto de su API para poder tener un conjunto de datos casi en *streaming*.

Sabiendo esto, vamos a imaginarnos el siguiente escenario. El gobierno belga quiere monitorizar el tráfico de sus carreteras y de sus sensores y para ello contacta con nosotros para que le construyamos la plataforma de datos con el objetivo de poder darle un servicio con el cual ellos puedan detectar si tienen sensores rotos o que puedan estar fallando y además poder detectar anomalías en el tráfico de sus carreteras. Les gustaría contar con un sistema lo más actualizado posible para poder reaccionar rápidamente a fallos en los sensores para arreglarlos. También quieren disponer de un sistema a tiempo real o lo más cercano a tiempo real para poder detectar anomalías en las carreteras del país (accidentes, atascos, posible necesidad de construir más carreteras para repartir el tráfico de una carretera entre varias, etc).

Además, el gobierno belga quiere alquilar espacio publicitario a empresas de marketing en sus carreteras. De esta forma, podrá detectar cuáles son las carreteras con más tráfico dependiendo de la hora y el día para poder ponerle precio. El precio de las diferentes carreteras es calculado en función de un algoritmo predictivo interno que tienen ellos. Lo que necesitan para hacer este algoritmo funcionar es la cantidad de tráfico total durante la última semana por cada tramo de sensor.

Por lo tanto, hay 3 necesidades a satisfacer:

- Estado de los sensores en tiempo real o casi.
- Estado de las carreteras en tiempo real o casi.
- Cantidad de personas que están pasando a una hora determinada por cada tramo de carretera.

Hoy en día aún es muy común ver todo lo relacionado con el Big Data y la computación en la Nube como algo difícil. De hecho, para muchos aún les viene a la cabeza cuando se lo nombras como que es algo solo para grandes empresas por los costes que esto acarrea.

En referencia a la computación a la nube hay muchos mitos que hacen a la gente confundirse. Si bien es cierto que el costo de mantener un servicio o software en la nube puede dispararse, son muchas las veces en las que una empresa emplea estos servicios y arquitecturas proporcionados por empresas como Microsfot Azure, Amazon Web Services o Google Cloud [3] por la baja barrera de entrada que supone para un negocio y porque el costo de mantenimiento de ese servicio es realmente bajo a si tuviera que ejecutar esos procesos en un centro de datos privado o en un ordenador local.

Cuando una empresa o una persona pretende desarrollar un servicio o producto, en el cuál va a tener que hacer uso de mucho poder de cómputo, va a tener que almacenar mucha información o simplemente alojar la aplicación en un lugar donde multitud de personas puedan acceder, muchas veces se plantea

como solución el hacer uso de un ordenador personal o un servidor comprado simple y exclusivamente para poder satisfacer esta tarea. Esto implica que la barrera de entrada aumente desde que el gasto inicial ya está basado en la máquina y recursos que necesitemos para hacer funcionar nuestro servicio. Estos gastos se denominan “gastos de capital” (*CapEx)*.

Sin embargo, si en lugar de llevar nuestra aplicación a nuestro propio servidor, lo desplegamos o lo desarrollamos basándonos en servicios que la nube puede darnos, nuestros gastos de capital se reducirían a 0. Por lo que los únicos gastos con los que tendríamos que contar serían con los “gastos operacionales” (*OpEx)*. Los gastos operacionales son los gastos relacionados con los servicios comprados o alquilados por una empresa o una persona a la hora de desarrollar su producto.

Es por esto, que uno de los objetivos a cumplir con el desarrollo de este proyecto es desmentir que la nube es un servicio caro y solo apto para medianas o grandes empresas.

Por otro lado, otro de los objetivos a conseguir es el hecho de poder dar a conocer arquitecturas y tecnologías muy útiles que para la gran mayoría son totalmente desconocidas. Hay muchos servicios ya desarrollados y ofrecidos por las empresas Cloud que nos permiten acelerar el desarrollo de un producto o servicio muchísimo.

1.3 Objetivos

Como objetivos ligados a la solución del problema a resolver, el primero de ellos está basado en construir un sistema que pueda ofrecer en tiempo real el estado de los sensores que hay instalados en las diferentes carreteras. Para una empresa, el tiempo de reacción para solucionar un problema con su infraestructura es crucial. Ser capaz de detectar este problema en cuestión de segundos o minutos puede marcar una diferencia entre hacer a una empresa perder miles de euros o hacer que la empresa pueda reaccionar eficazmente para poner una solución.

En segundo lugar, un segundo objetivo relacionado con el problema a resolver es el hecho de poder saber en tiempo real el estado de una carretera. Esto es también algo crucial en la solución ya que poder identificar anomalías en una carretera puede ayudar a tomar diferentes decisiones. Podríamos identificar si una carretera está recibiendo más tráfico del habitual o menos y a raíz de ahí sacar conclusiones para poder actuar. Gracias a esto se podrían identificar atascos, accidentes, necesidad de nuevas carreteras, etc.

Por último, el tercer objetivo relacionado con el problema a resolver es dar la posibilidad de tomar decisiones respecto a la cantidad de personas que circulan a una determinada hora por una vía. Con esto se podrán estipular unos precios a los paneles de publicidad.

1.4 Planificación del proyecto

Para la elaboración y planificación del proyecto he hecho uso de una herramienta que integra Azure llamada DevOps. Una de las peculiaridades de esta herramienta es que cuenta con un apartado llamado "*boards*" o tableros. Aquí, podemos llevar a cabo el seguimiento de nuestro proyecto con el apoyo que hace a las metodologías ágiles. Podemos ver en la Figura 1 todas las historias de usuario que describían el proyecto e internamente a cada historia de usuario las tareas para conseguir completar esa historia de usuario.

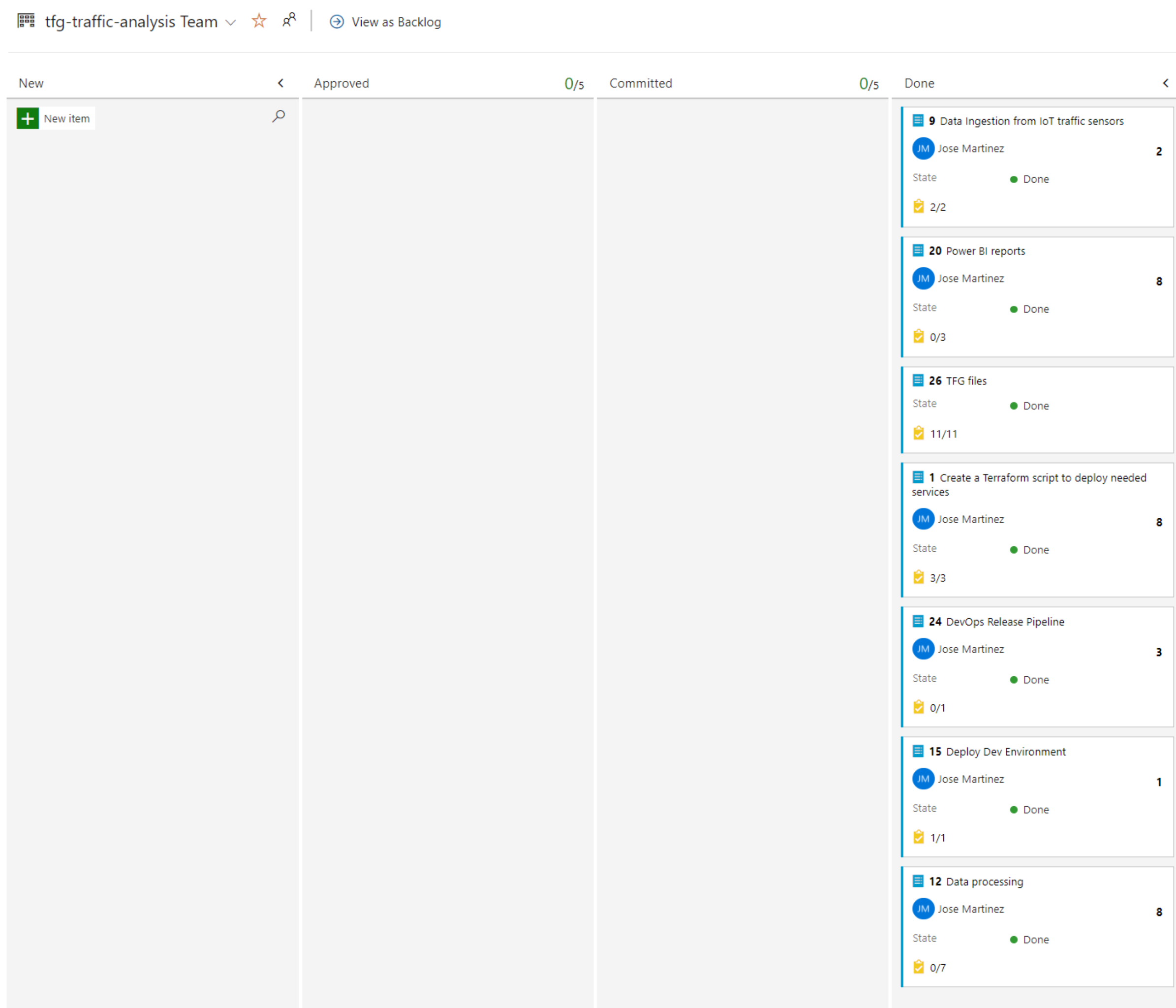


Figura 1. Tareas completadas en DevOps.

El desarrollo del proyecto se ha llevado a cabo en diferentes fases:

- FASE 1: La primera fase trata de analizar y entender las fuentes de datos con las que se iban a trabajar. Es importante tener un buen conocimiento de los datos sobre los que vamos a trabajar para poder alcanzar mejores resultados en nuestro sistema.
- FASE 2: Durante esta fase comenzamos a recolectar los datos. Construimos el sistema por el cual podemos obtener las fuentes de datos y almacenarlas en un primer nivel en nuestro sistema.

- FASE 3: En esta fase, realizamos la parte del procesamiento de los datos. Esta fase constaba de dos partes. Una primera para el procesamiento de los datos en streaming y una segunda para el procesamiento de los datos diarios.

- FASE 4: Aquí, nos encargamos de la elaboración de los diferentes informes. A partir de los datos procesados construimos los informes de datos para mostrar los resultados.

- FASE 5: Durante esta fase preparamos los scripts de infraestructura como código para poder desplegar la infraestructura rápidamente, agilizando así la incorporación de nuevos desarrollos al sistema.

- FASE 6: En esta fase se construyen los pipelines de CI/CD, en los cuales construiremos la solución cada vez que tengamos que integrar una nueva característica y además desplegamos la solución.

- FASE 7: Por último, desarrollamos la memoria del sistema explicando las diferentes tecnologías usadas para desarrollar el proyecto y cómo funciona en sí.

El desarrollo finalmente se llevó a cabo en 3 meses (Fase 1 a 6). Los 3 meses se dividieron en 8 sprints de semana y media cada uno, quedándose una división de las fases como se puede ver en la Figura 2.

Fases de desarrollo	Temporización del proyecto							
	1º	2º	3º	4º	5º	6º	7º	8º
Fase 1: Análisis de los feeds que vamos a procesar	■							
Fase 2: Desarrollo del sistema de ingestión de datos		■	■					
Fase 3: Desarrollo del sistema de procesamiento de datos			■	■				
Fase 4: Desarrollo del reporte del sistema					■	■		
Fase 5: Preparación del entorno de desarrollo							■	
Fase 6: Construir pipelines de CI/CD								■

Figura 2. Planificación final de sprints.

Durante cada sprint se llevó a cabo una carga de trabajo de 30 horas, excepto para el tercero, que contenía carga de trabajo tanto de la fase 2 como 3, para poder sacar adelante todas las tareas a realizar, tuve que hacer un sobre esfuerzo de 8 horas.

Por tanto, para el desarrollo del proyecto, se han empleado 8 sprint * 30 horas = 240 horas que sumadas a las 8 horas de esfuerzo extra en el sprint 3 hacen un total de 248 horas para el desarrollo del proyecto.

Por último, el desarrollo de la memoria del proyecto, en la fase 7, ha tenido una duración de lo que serían 2 sprints, a una carga de trabajo de 35 horas por sprint.

En total, el proyecto completo ha tenido una duración de 318 horas. El proyecto estaba estimado para ser de unas 300 horas, por lo que he tenido una diferencia de 18 horas de más empleadas a las planificadas. Esto viene a consecuencia de problemas hallados en el desarrollo durante la fase 4, en la realización de los informes de resultados, así como también en la realización de la memoria, a la cuál he tenido que dedicarle un tiempo superior al esperado.

1.5 Estructura del documento

La documentación del Trabajo de Fin de Estudios se ha estructurado de la siguiente manera:

- **Capítulo 1: Introducción.** En este capítulo es donde se describe brevemente en qué consiste el proyecto, cuáles son las motivaciones que han llevado a desarrollarlo y qué objetivos se pretenden alcanzar con su desarrollo.

- **Capítulo 2: Estado del arte.** En este capítulo se enumeran algunas aplicaciones que tienen un objetivo similar a la plataforma que se ha desarrollado. Se comentan similitudes y diferencias entre los diferentes sistemas.

- **Capítulo 3: Entendiendo el entorno del problema.** En este capítulo se realiza un análisis en profundidad del problema a solucionar. Se describen las diferentes entradas de datos y se plantea una solución.

- **Capítulo 4: Tecnologías, herramientas y servicios.** En este capítulo se realiza una explicación de los diferentes recursos utilizados para el desarrollo de la plataforma de datos en la nube de Microsoft Azure.

- **Capítulo 5: Arquitectura Delta.** En este capítulo se explica cuál es la arquitectura que se va a utilizar para desarrollar el producto. Se compara con otras arquitecturas de datos y se exponen las ventajas y desventajas de usar esta arquitectura frente al resto.

- **Capítulo 6: Traffic Monitoring. La plataforma de datos.** En este capítulo se explica cómo se ha construido y cómo funciona cada parte del producto que se ha construido para solucionar el problema. Aquí se exponen diferentes detalles de la implementación.

- **Capítulo 7: Resultados. Presentando los datos procesados en Power BI.** En este capítulo se muestran los reportes de donde se podrá extraer información relevante para la toma de decisiones. Además, se explica cómo hacer uso de cada reporte para saber cómo usarlo y poder extraer conclusiones útiles de ellos.

- **Capítulo 8: Conclusiones y trabajo futuro.** En este capítulo se expone qué conclusiones se han podido sacar del proyecto al desarrollar este producto, así como puntos de mejora del producto para futuros desarrollos.

- **Bibliografía.** Se citan todas las referencias y bibliografía que se han considerado para la realización del TFG.

2. Estado del arte

En la actualidad hay multitud de aplicaciones que ayudan a ver el estado del tráfico en cualquier momento del día o a detectar accidentes de tráfico. Todas ellas ofrecen unos servicios similares para facilitar a una persona a decidir si tomar una ruta u otra. Entre estas aplicaciones están por ejemplo Google Maps y Waze.

Luego están también aplicaciones de diseño de empresas tanto públicas como privadas para llevar a cabo estos mismos análisis, pero con objetivos más estadísticos y de apoyo al desarrollo de las carreteras. Este es el caso por ejemplo del sistema que utiliza la DGT para controlar esto.

Por otro lado, para el caso de la publicidad y cálculo de impactos y visualizaciones de un anuncio existen productos como Route API de la empresa Mediatel.
Vamos a ver un poco más en profundidad estos productos.

2.1 Waze

Waze es una aplicación móvil que se integra con tu vehículo y te avisa de carreteras cortadas, controles de tráfico, etc. Además, cuenta también con integración con aplicaciones de música de forma que puedes controlar la música que escuchas en el coche fácilmente (ver Figura 3).

Waze también se encarga de llevarte por la ruta más corta, recomendando el mejor trayecto para ahorrar tiempo y combustible en nuestro vehículo. De esta forma, Waze nos ayuda a evitar atascos accidentes en la vía.

Otra de las propiedades que tiene esta aplicación es que nos ayuda a buscar aparcamiento cuando estemos llegando a nuestro destino.

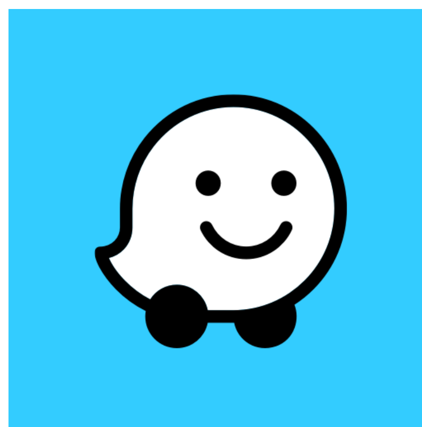


Figura 3. Logotipo de Waze.

2.2 Google Maps

Una de las aplicaciones estrella de la empresa multinacional Google es la aplicación Google Maps. Esta aplicación móvil, aparte de ser una de las más utilizadas cada día para saber cómo ir de un lugar a otro, también puede ser utilizada para consultar el estado del tráfico en tiempo real en nuestra zona o en una ruta en concreto.

En la APP contamos con diversos tipos de visualizaciones. Entre ellas tenemos la visualización de "Tráfico" (ver Figura 4).

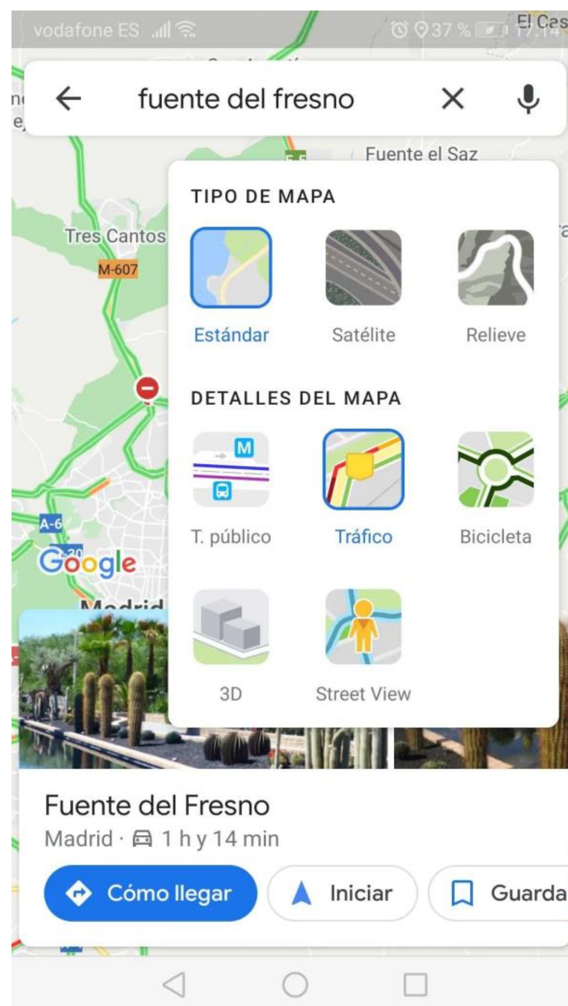


Figura 4. Selección de vista modo Tráfico.

Cuando seleccionamos esta vista, podremos ver sobre el típico mapa de la aplicación como de fluido es el tráfico sobre la vía, marcados con colores verde, naranja y rojo, de más a menos fluidez.

Además,de igual manera que Waze, esta aplicación también nos dirá cual es el mejor trayecto para ir de un punto A a un punto B.

2.3 Sistemas de la Dirección General de Tráfico

La Dirección General de Tráfico (DGT) cuenta también con sus propios sistemas internos para el control del tráfico en las carreteras españolas. Este sistema cuenta con sensores (ver Figura 5) para monitorear la cantidad de tráfico, la velocidad de los vehículos, la distancia entre vehículos, etc.

Este sistema interno, mucho menos visual que el de las otras dos aplicaciones, otorga información muy importante a la hora de tomar decisiones sobre si hay demasiado tráfico en una zona y poder ver cuál es su motivo.

Entre otros tipos de medidas que también controlan los sistemas de la DGT son los controles de velocidad con radares fijos y de tramo.

Figura 5. Imagen de sensor DGT.

2.4 Mediatel. Route API

Route es un producto de la empresa Mediatel que consiste en el cálculo de la audiencia estimada que podrá ver un anuncio en un panel de publicidad. Además de cuantas personas verán un anuncio, también calculan con qué frecuencia verán ese panel. (ver Figura 6)

Está información la venden a empresas para que estas puedan planificar, comprar, vender y evaluar una inversión en publicidad en un punto.

Figura 6. Logotipo de Route

2.5 Conclusiones

Con los ejemplos anteriores hemos podido ver algunos de los casos de uso que se le dan a los datos que vamos a manejar y que utilidad tienen en el mundo real. Unas aplicaciones están más orientadas a un público más genérico como puede ser el caso de Google Maps o Waze y luego está el caso opuesto como es el sistema de la DGT.

Para nuestro sistema, buscaremos algo intermedio. Un sistema que nos permita detectar anomalías en las diferentes carreteras, así como mostrar información para poder tomar decisiones en base a esos datos.

Además, nuestro producto contará con un sistema para poder calcular la audiencia que ha circulado de forma horaria por cada vía. De esta forma podremos dar una estimación de personas por vía y por hora para que la empresa lo use y pueda calcular sus precios.

Característica	Waze	Maps	DGT	Route	Traffic Monitoring
Ver estado del tráfico en las carreteras.	Si	Si	Si	No	Si
Ver cantidad de vehículos en un tramo.	No	No	Si	No	Si
Sugerencia de rutas alternativas.	Si	Si	No	No	No
Estimaciones sobre afluencia de gente.	No	No	No	Si	Si
Aumento de tráfico en el tiempo.	No	No	NS/NC	No	Si
Identificación de irregularidades en las carreteras.	Si	Si	Si	No	Si

3. Entendiendo el entorno del problema

Una vez que sabemos que es lo que queremos hacer y qué es lo que tenemos que conseguir, vamos a pasar a comprender mejor el problema. Vamos a comenzar por entender completamente el problema a resolver.

Para ello vamos a necesitar primero examinar los datos que tenemos de entrada para saber con qué información contamos y cuál vamos a tener a nuestra disposición para poder afrontar una solución.

Por último, una vez comprendidos los recursos de los que disponemos y el objetivo final a conseguir, podremos plantear una solución al problema.

3.1 Analizando el problema

El gobierno belga ha contactado con nosotros para desarrollarles una plataforma de datos que les permita tener centralizada y relacionada la información que recogen de los diferentes sensores que tienen instalados en sus carreteras.

Con estos datos quieren que consigamos darle la siguiente información, para que ellos puedan, en función de los resultados que les proporcionemos a través del procesamiento de sus datos, tomar una decisión u otra.

En primer lugar, quieren que podamos desarrollar un sistema, que les permita saber en cualquier momento del día el estado de todos y cada uno de los sensores instalados. Con esto serán capaces de monitorizar sus sistemas, para poder saber si están funcionando o no correctamente. En caso de que no estén funcionando correctamente se podrá enviar un equipo técnico para arreglar cualquiera que sea el problema que tenga el sensor.

Para lograr este primer requisito, necesitaremos de un sistema en tiempo real que esté continuamente procesando la información que recibimos. Los datos de los sensores nos llegarán una vez por minuto. Aunque esto no significa que haya información más actualizada, necesitaremos procesar los archivos para ver los datos que contienen y poder actualizar la información del estado de los sensores una vez lleguen cambios en los archivos.

El criterio de aceptación para dar este requisito como completado será tener un informe donde se pueda ver a cualquier hora del día cuál es el estado de los sensores. Se ha de poder ver de forma clara y concisa el estado de un sensor, así como la localización de un sensor para poder enviar al equipo técnico a arreglarlo en caso de no funcionar correctamente.

En segundo lugar, quieren también que, con la información que proporcionan los sensores, demos unas métricas del estado del tráfico. Con esto se podrán detectar anomalías en las carreteras belgas para intentar detectar accidentes, atascos, necesidad de nuevas vías para diversificar el tráfico en una zona, etc. Para esta parte necesitaremos tener en cuenta dos factores. Por un lado necesitaremos tener un histórico de datos del tráfico que circula por una carretera durante un día a una hora determinada. Después tendremos que calcular en tiempo real la cantidad de tráfico que circula por una carretera durante la hora presente. Con estos últimos datos podremos comparar con el histórico de datos que tenemos, que necesitará irse actualizando con el tiempo, para poder detectar si el tráfico a una hora determinada durante un día de la semana determinado es mayor o menor que la media que suele presentar esa carretera en cuestión.

El criterio de aceptación para dar como completado este requisito será el poder ver en un informe cuales son las carreteras que cuentan con un tráfico mayor del esperado, cuáles son las que tienen un tráfico menor del esperado y cuáles el esperado, así como la localización de estos tramos.

Por último, tendremos que, con la información que nos proporcionan los sensores realizar un cálculo de cuántas personas están circulando por una vía a una determinada hora para la venta de espacio publicitario. Una carretera con una afluencia de gente mayor tendrá un mayor precio que una con una afluencia menor ya que un anuncio en esa vía tendrá una mayor cantidad de impactos en personas que la otra. Este requisito no hará falta que sea en tiempo real, y se podrá procesar esta información de forma diaria, de forma que los precios de los anuncios cambiarían diariamente. Nuestro sistema solo tendrá que proporcionar la afluencia de personas y luego sus sistemas internos ya se encargará de calcular los precios en función de los datos que les enviemos.

El criterio de aceptación para dar como completado este requisito será el poder ver en un informe el número de personas que circulan por una vía a una determinada hora durante un día.

3.2 Fuentes de datos

Los sensores de las carreteras belgas tienen un período de actualización estimado de alrededor de 1 minuto. Sin embargo, no todos los sensores enviarán una actualización a cada minuto. Se puede dar la posibilidad de que un sensor se actualiza cada 2 o 3 minutos.

Los sensores envían los datos a una Interfaz de Programación de Aplicaciones (API). Será aquí donde tendremos que hacer una solicitud para recuperar los datos de los sensores.

El gobierno belga nos ha facilitado las direcciones de la API a las que tendremos que llamar para conseguir los datos. Tendremos 2 direcciones a las que llamar para conseguir los datos.

Vamos a ver las diferentes fuentes de datos más en profundidad.

3.2.1 Listado de sensores

Esta fuente de datos es la que nos proporcionará información relacionada con el sensor. Tendremos que llamar a la siguiente dirección para obtener esta información:

"http://miv.opendata.belfla.be/miv/configuratie/xml" [1]

Las actualizaciones que recibe esta fuente de datos son mínimas. Aquí podremos encontrar un listado de todos los sensores e información relevante al sensor. Ya que la mayoría de los puntos clave de las carreteras ya han sido cubiertos y los sensores están ya instalados, rara vez tendremos alguna actualización en estos datos. Aun así, tendremos que conseguir estos datos para ver cuando se instala un nuevo sensor y poder así empezar a procesar los datos de ese sensor.

Vamos a ver un ejemplo de respuesta de la API. (ver Figura 7).


```
<tijd_laatste_config_wijziging>2019-12-05T10:26:49+01:00</tijd_laatste_config_wijziging>
▼<meetpunt unieke_id="3640">
   <beschrijvende_id>H291L10</beschrijvende_id>
   <volledige_naam>Parking Kruibeke</volledige_naam>
   <Ident_8>A0140002</Ident_8>
   <lve_nr>437</lve_nr>
   <Kmp_Rsys>94,695</Kmp_Rsys>
   <Rijstrook>R10</Rijstrook>
   <X_coord_EPSG_31370>144474,5297</X_coord_EPSG_31370>
   <Y_coord_EPSG_31370>208293,5324</Y_coord_EPSG_31370>
   <lengtegraad_EPSG_4326>4,289731136</lengtegraad_EPSG_4326>
   <breedtegraad_EPSG_4326>51,18460764</breedtegraad_EPSG_4326>
</meetpunt>
```


Figura 7. Ejemplo de respuesta API. Descripción del sensor.

Como podemos ver, los nombres de los campos vienen en belga. Pero podemos encontrar la documentación de la API en inglés en la siguiente dirección web:

"http://miv.opendata.belfla.be/miv-config.xsd" [2]

Entre los campos más importantes de la respuesta de la API tenemos los siguientes:

- tijd_laatste_config_wijziging: Nos dice la fecha y hora de cuando se generó el actual archivo de configuración.
- unieke_id: Número de identificación del punto de medida.
- beschrijvende_id: Identificador descriptivo. Este identificador se usa internamente.
- lve_nr: Identificador de la unidad de procesamiento.
- Rijstrook: Hace referencia al carril del punto de medida. El carácter indica el tipo de línea.
- lengtegraad_EPSG_4326: Longitud acorde a WGS84.
- breedtegraad_EPSG_4326: Latitud acorde a WGS84

3.2.2 Información de los sensores

Esta fuente de datos es la que nos proporcionará los datos recogidos y medidos por el sensor. Tendremos que llamar a la siguiente dirección para obtener esta información:

"http://miv.opendata.belfla.be/miv/verkeersdata" [3]

Esta fuente de datos se actualiza con bastante frecuencia teniendo una actualización a casi cada minuto. Como con cada llamada guardamos la última versión de los datos necesitamos guardar una versión de los datos en el menor rango de actualización posible, en este caso por minuto. Estos datos serán cruciales para la construcción de los informes y reportes que necesitamos hacer. Estos archivos son los que contienen toda la información del sensor, tales como frecuencia de actualización, conexión, mediciones, etc.

Vamos a ver un ejemplo de respuesta de la API. (ver Figura 8).


```
<tijd_publicatie>2020-08-24T08:48:54.04+01:00</tijd_publicatie>
<tijd_laatste_config_wijziging>2019-12-05T10:26:49+01:00</tijd_laatste_config_wijziging>
▼<meetpunt beschrijvende_id="H222L10" unieke_id="29">
  <lve_nr>55</lve_nr>
  <tijd_waarneming>2020-08-24T08:47:00+01:00</tijd_waarneming>
  <tijd_laatst_gewijzigd>2020-08-24T08:48:28+01:00</tijd_laatst_gewijzigd>
  <actueel_publicatie>1</actueel_publicatie>
  <beschikbaar>1</beschikbaar>
  <defect>0</defect>
  <geldig>0</geldig>
 ▼<meetdata klasse_id="1">
    <verkeersintensiteit>0</verkeersintensiteit>
    <voertuigsnelheid_rekenkundig>0</voertuigsnelheid_rekenkundig>
    <voertuigsnelheid_harmonisch>252</voertuigsnelheid_harmonisch>
  </meetdata>
 ▼<meetdata klasse_id="2">
    <verkeersintensiteit>0</verkeersintensiteit>
    <voertuigsnelheid_rekenkundig>0</voertuigsnelheid_rekenkundig>
    <voertuigsnelheid_harmonisch>252</voertuigsnelheid_harmonisch>
  </meetdata>
 ▼<meetdata klasse_id="3">
    <verkeersintensiteit>0</verkeersintensiteit>
    <voertuigsnelheid_rekenkundig>0</voertuigsnelheid_rekenkundig>
    <voertuigsnelheid_harmonisch>252</voertuigsnelheid_harmonisch>
  </meetdata>
 ▼<meetdata klasse_id="4">
    <verkeersintensiteit>0</verkeersintensiteit>
    <voertuigsnelheid_rekenkundig>0</voertuigsnelheid_rekenkundig>
    <voertuigsnelheid_harmonisch>252</voertuigsnelheid_harmonisch>
  </meetdata>
 ▼<meetdata klasse_id="5">
    <verkeersintensiteit>0</verkeersintensiteit>
    <voertuigsnelheid_rekenkundig>0</voertuigsnelheid_rekenkundig>
    <voertuigsnelheid_harmonisch>252</voertuigsnelheid_harmonisch>
  </meetdata>
 ▼<rekendata>
    <bezettingsgraad>0</bezettingsgraad>
    <beschikbaarheidsgraad>100</beschikbaarheidsgraad>
    <onrustigheid>0</onrustigheid>
  </rekendata>
</meetpunt>
```


Figura 8. Ejemplo de respuesta API. Mediciones del sensor.

Al igual que en los datos anteriores, estos también vienen en belga. Pero podemos encontrar la documentación de la API en inglés en la siguiente dirección web:

“http://miv.opendata.belfla.be/miv-verkeersdata.xsd” [4]

Entre los campos más importantes de las respuestas de la API tenemos los siguientes:

- tijd_publicatie: Es la fecha y hora de cuando fueron generados los datos.
- tijd_laatste_config_wijziging: Es la fecha y hora de la última actualización de los datos de configuración.
- unieke_id: Número de identificación del punto de medida.
- beschrijvende_id: Identificador descriptivo. Este identificador se usa internamente.
- tijd_laatst_gewijzigd: Es la fecha de última actualización del punto de medida.
- beschikbaar: Muestra la disponibilidad del punto de medida. 0 = No disponible. 1 = Disponible.
- defect: Muestra si el sensor tiene fallos. 0 = Sin fallos. 1 = Posibles fallos. 2 = +20% de datos incorrectos.
- geldig: Muestra la regularidad de los datos. 0 = Regular. 1 = Irregular. 2 = Extremadamente irregular. 3 = Extremadamente irregular debido a fallos.
- klasse_id: Identificador del tipo de vehículo en la medida. Hay 5 tipos de vehículos.
- verkeersintensiteit: Contador de vehículos por tipo.

3.3 Planificando la solución

Ahora ya estamos preparados para presentar una solución. Tenemos conocimiento de los recursos de los que disponemos y sabemos cuáles son nuestras metas a conseguir.

En primer lugar, vamos a tener en cuenta las diferencias a la hora de desarrollar. Con esto quiero decir que hay partes de nuestro desarrollo que necesitarán ser en tiempo real por lo que necesitaremos hacer uso de tecnologías y servicios que se presten a ello, otorgándonos unos niveles de latencia bajos para tener el mínimo tiempo de respuesta posible y así disponer de los datos procesados y listos para ser mostrados en nuestros informes en la menor cantidad de tiempo posible.

Basándonos en esto, vamos a necesitar tener un sistema de ingestión de datos con el que podamos planificar una llamada a la API a cada minuto para cada fuente de datos. Como vamos a desarrollar la solución en Microsoft Azure [5] (ver Figura 9), vamos a basarnos en los servicios que esta plataforma nos ofrece.

Figura 9. Logotipo de Microsoft Azure.

Para la parte de la ingestión de los datos, hay 3 servicios que pueden adaptarse a nuestras necesidades:

- Azure Data Factory [6]
- Azure Function APP [7]
- Azure Logic APP [8]

La opción de Azure Logic App, la vamos a descartar por ser más complicada para comprimir los datos conseguidos de la API a la hora de guardarlos. Por lo tanto, nos quedaremos con la opción de Azure Data Factory (ver Sección 4.3.2) y Azure Functions (ver Sección 4.3.5). Entre estas dos opciones, vamos a descartar la opción de Azure Data Factory por no poder programar de una manera sencilla las llamadas a la API a cada minuto. Por tanto, nuestra mejor opción sería quedarnos con el servicio de Azure Functions App para realizar las llamadas a la API.

Para el almacenamiento de los datos vamos a utilizar el servicio de Storage account [9] (ver Sección 4.3.8). En este servicio será donde daremos forma a nuestro Data Lake [10, 45].

Entonces, vamos a descargar los datos desde nuestra *Azure Function* y guardaremos los datos en el Storage Account. Para poder llevar un log de las llamadas que haremos a la API haremos uso de uno de los subservicios que tiene Storage Account, Table Storage [11] (ver Sección 4.3.8).

Ahora que ya tenemos planificado cómo será la ingestión de los datos, vamos a comenzar con el procesamiento de estos. Tendremos que dividir esta parte en dos caminos diferentes. Por un lado, tendremos la vía de procesamiento rápida, donde procesaremos la cantidad mínima de datos posible para satisfacer las necesidades para los reportes de tiempo real. Esto lo haremos así para abaratar costes, ya que el procesamiento en tiempo real durante mucho tiempo podría ser caro.

Para el procesamiento de los datos en *streaming* vamos a usar el servicio *Streaming Analytics* [12, 44] (ver Sección 4.3.7). Este servicio está pensado para usarse en *Streaming* de medidas tomadas desde sensores, estando conectado directamente con los sensores. Nosotros no podemos tener este servicio conectado directamente con los sensores, pero sí que podemos conectarlo con el Data Lake y procesar los archivos en cuanto lleguen. Al ser la rama en tiempo real, una de las características que mejor nos vendrán de este servicio es que solo guarda datos que hayan llegado hasta una hora antes, así que contaremos siempre con los datos de la hora anterior. Los datos procesados aquí los enviaremos directamente a un conjunto de datos a la herramienta que vamos a usar para hacer los informes, Power BI [13] (ver Sección 4.2.8).

Por la otra rama tendremos nuestro procesamiento diario. Esta parte se ejecutará una vez al día y procesa todos los datos que hayan llegado durante el día. para esto si que podremos utilizar el Data Factory. Desde aquí ejecutaremos las actividades de procesamiento para que lancen o llamen a los servicios que necesite. Haremos uso de un Tumbling Window [14] (ver Sección 4.3.2) para lanzar las actividades diarias.

Además, para evitar carga de trabajo extra en la lógica de procesamiento, lo que haremos será utilizar el apoyo de otros dos servicios de Azure. Estos servicios son Azure Event Grid [15] (ver Sección 4.3.9) y Azure Storage Queue [16] (ver Sección 4.3.8). El hecho de usar estos servicios es aprovechar una de las ventajas de usar un Storage Account para almacenar nuestros datos. Cuando un nuevo archivo llega

a nuestro Data Lake este genera diferentes eventos en función de cuando se crea el archivo, se escribe, se termina de escribir o se elimina, entre otros eventos. Con estos servicios lo que haremos será reaccionar al evento de terminar de escribir un archivo de forma que cuando nuestra Azure Function escriba un nuevo bloque de datos, el servicio de Event Grid detectará que ha ocurrido un evento. Al reaccionar a este nuevo evento, lo que haremos es, enviar a nuestro otro servicio, el Storage Queue, un mensaje donde le diremos la dirección en el Data lake de este nuevo archivo que ha llegado.

De esta forma, al finalizar el día, cuando ejecutemos nuestra actividad de procesamiento diario, junto con otro servicio llamado Azure Databricks [17] (ver Sección 4.3.4), podremos leer solo los archivos que han llegado nuevos durante este día. Para ello en Databricks, cuando ejecutemos nuestro proceso de lectura, utilizaremos una tecnología llamada ABS-AQS. Con esto lo que haremos será leer todos los mensajes que metimos en la cola del Storage Queue y leeremos del Data Lake todos los archivos listados. Una vez leídos los archivos, nuestra cola se quedará limpia y preparada para el día siguiente.

Otra ventaja de usar este método es que Spark [18, 42] (ver Sección 4.2.6), otra tecnología que usaremos para procesar los archivos que está orientado al procesamiento de grandes cantidades de datos, cuando le dices que lea por ejemplo un día entero de datos y esa carpeta tiene múltiples directorios, Spark consumirá tiempo y recursos en listar los archivos, más que en leerlos directamente si le damos todos los directorios exactos de cada archivo.

Cuando acabemos de procesar los datos, los guardaremos en otro servicio de Azure llamada SQL Database [19] (ver Sección 4.3.3). Esta base de datos tiene la peculiaridad de ser bajo demanda, por lo que, al igual que sucede con las Azure Functions, solo pagaremos por el uso que le estemos dando.

Por último, para mantener un nivel de seguridad en nuestro sistema, vamos a hacer uso del servicio Key Vault [29] (ver Sección 4.3.6) para almacenar nuestras conexiones y contraseñas que necesitarán ser usadas por los diferentes servicios que utilizaremos.

*Datalake: Repositorio centralizado que te permite almacenar todos los datos ya sean estructurados y no estructurados a cualquier escala.

Podemos ver como quedaría nuestra arquitectura en la Figura 10.

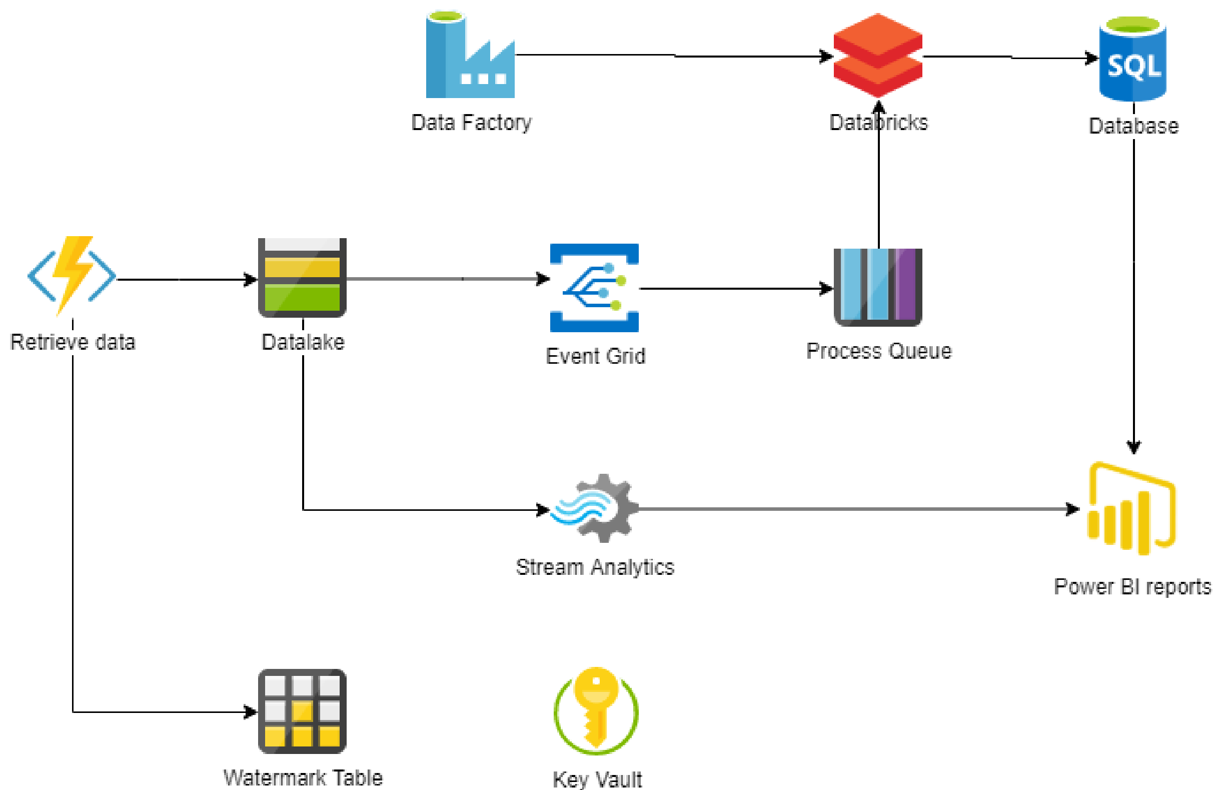


Figura 10. Esquema de arquitectura a construir.

Ahora que hemos visto cuál sería la idea de arquitectura a construir, vamos a pasar a la implementación. La mayor parte de la implementación vendrá en Databricks, donde tendremos que implementar la mayor carga de procesamiento de nuestro sistema.

Utilizaremos el lenguaje de programación Python [20][21] para construir la solución apoyándonos de la librería de Spark para Python, Pyspark.

Además, para construir el Data Lake vamos a utilizar una tecnología llamada Delta lake [22] (ver Sección 4.2.9). Construiremos varias capas de datos donde iremos transformando nuestros datos. De esta forma tendremos una capa denominada Raw donde llegarán los archivos tal cual son obtenidos de la API. Luego tendremos una capa de Landing/Bronze, donde tendremos los datos de Raw sin transformación ninguna pero almacenados en un Delta. Después transformaremos los datos, limpiandolos y dejándolos listos para ser utilizados en una capa posterior, teniendo aquí nuestra capa de Staging/Silver. Por último, tendremos nuestra capa de agregación que es donde tendremos nuestros datos calculados y listos para guardar a la base de datos o ser usados en el reporte. Esta capa se denomina Curated/Gold.

De esta forma, construiremos la arquitectura en la que se basará todo nuestro sistema, llamada Arquitectura Delta (ver Capítulo 5). (ver Figura 11).

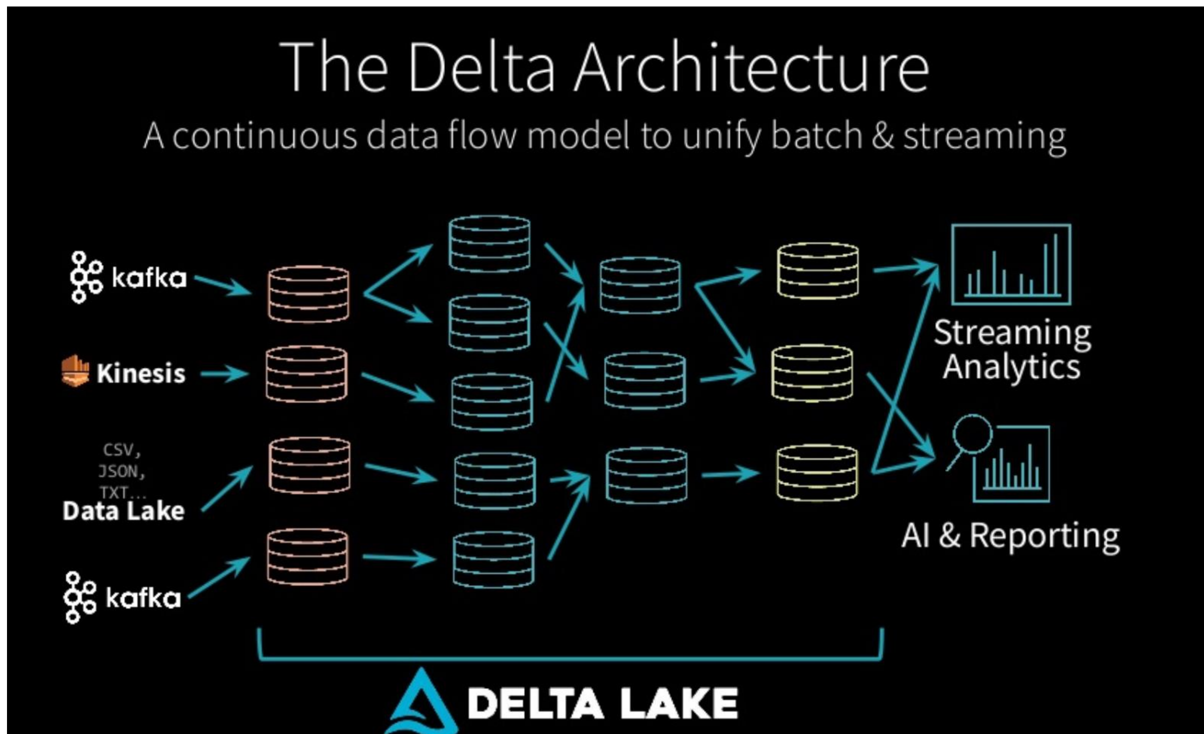


Figura 11. Diagrama de Arquitectura Delta. [22]

Por último, además de utilizar la arquitectura Delta y Spark, vamos a utilizar Structured Streaming [23] (ver Sección 4.2.7). Structured Streaming es un motor de procesamiento en *streaming*, que permite también el procesamiento en *batch* de forma que le daremos mucha versatilidad a todo nuestro sistema, construyendo un sistema robusto y abierto a mejoras en un futuro.

4. Tecnologías, herramientas y servicios

En este capítulo vamos a ver en profundidad cada una de las tecnologías, herramientas y servicios que se usan para el desarrollo del sistema a construir.

4.1 Tecnologías

En este apartado nos vamos a centrar solamente en las tecnologías usadas para la implementación de la lógica de nuestro sistema.

4.1.1 Python

Python es un lenguaje de programación multiparadigma utilizado para programación orientada a objetos, la programación imperativa y a programación funcional. Se trata de un lenguaje de código abierto que es administrado y mantenido por la Python Software Foundation. (ver Figura 12)

Python es uno de los lenguajes de programación más simples y fáciles de entender. Tiene una sintaxis sencilla que hace que su curva de aprendizaje sea bastante asequible para iniciados a este lenguaje.

Unas de las características más destacables de Python es la posibilidad de crear un código que cuenta con una gran legibilidad, ahorrando bastante tiempo y recursos.

Es muy común utilizar Python para el desarrollo de aplicaciones para el análisis de datos y la extracción de información útil en Big Data. Además, cuenta con librerías que apoyan este objetivo como bien es la librería de Pandas, Pydoop o Pyspark.

Python es un lenguaje interpretado, que junto con su tipado dinámico lo convierten en un lenguaje de programación perfecto para scripting y desarrollo rápido de aplicaciones.

Figura 12. Logotipo de Python.

4.1.2 Transact-SQL

Transact-SQL [24] (T-SQL) es una extensión que realizó Microsoft a SQL (*Structured Query Language)*. T-SQL es fundamental a la hora de trabajar con servicios de Microsoft como pueden ser SQL Server u otros servicios de bases de datos de Microsoft.

Técnicamente cuando aprendes a usar T-SQL también estás aprendiendo a usar SQL, y viceversa, ya que el primero es una extensión del segundo.

Por tanto, T-SQL se trata de un lenguaje de consultas de bases de datos diseñado para administrar y recuperar información de sistemas de bases de datos relacionales. (ver Figura 13).

Figura 13. Logotipo de T-SQL.

4.2 Herramientas

En este apartado nos vamos a centrar solamente en las herramientas que se han usado para desarrollar el sistema.

4.2.1 Visual Studio Code

Visual Studio Code (VS Code) (ver Figura 14) es un editor de código desarrollado por Microsoft. Tiene soporte para la depuración e integración con Git.

Uno de los puntos más destacables de VS Code es la comunidad que tiene detrás. Esta comunidad apoya este editor desarrollando multitud de extensiones muy útiles que ayuden con el desarrollo y prueba de nuestras aplicaciones.

Gracias a esto, VS Code tiene soporte para el desarrollo en multitud de lenguajes de programación. Es completamente gratuito y de código abierto.

Además, respecto a nuestro desarrollo tiene la peculiaridad de que cuenta con integración directa con Azure, por lo que podemos realizar despliegues directamente desde el editor de código o acceder a recursos a través de él.

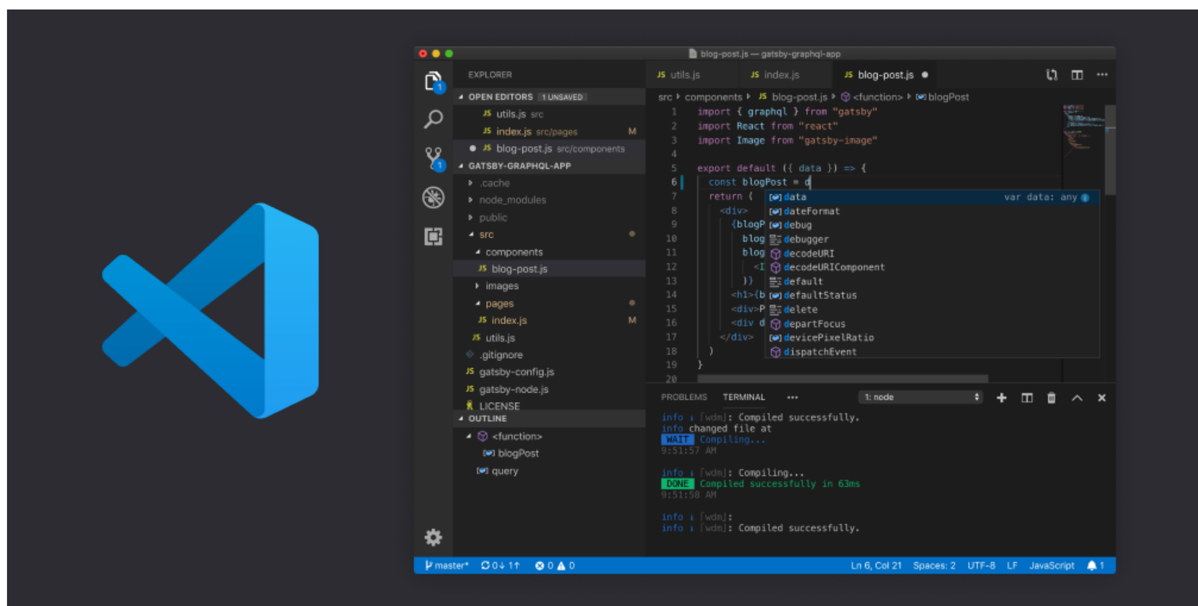


Figura 14. Interfaz y logotipo de VS Code.

4.2.2 Azure

Microsoft Azure es un conjunto de servicios que ofrece Microsoft en la nube con el objetivo de ayudar a otras organizaciones o personas a satisfacer sus necesidades.

Azure permite el desarrollo de soluciones en la nube pública completamente o soluciones híbridas. A veces las empresas buscan expandir sus recursos pero manteniendo sus servidores privados. Es muy común encontrarse con soluciones híbridas bajo estas condiciones.

Azure cuenta con multitud de servicios que pueden ser desplegados en múltiples regiones. Existen regiones que están reservadas para ciertas instituciones como es por ejemplo la región de Azure Government, que está reservada para instituciones y centros federales de los Estados Unidos, así como *contractors* del gobierno estadounidense.

Los recursos son desplegados en Azure en Grupos de Recursos. Un grupo de recursos es una unidad lógica dentro de nuestra subscripción que nos permitirá mantener una organización.

Dentro de Azure, normalmente vamos a pagar por aquello que utilicemos aunque también se puede pagar por adelantado por recursos a utilizar (lo cuál sale más económico).

4.2.3 Git

Git es un software de control de versiones (ver Figura 15) que fue creado pensando en la eficiencia y la confiabilidad del mantenimiento de versiones de aplicaciones así como pensando en el desarrollo de aplicaciones llevado a cabo por multitud de personas simultáneamente.

Algunas de las características más importantes de Git son:

- Gestión de ramas para dividir el desarrollo y luego poder volver a juntarlas conforme vamos acabando características.
- Es software libre.
- Cada rama puede tener una línea de progreso diferente de la rama principal.
- Podemos mantener un historial completo de versiones y volver a una versión anterior en caso necesario.

Figura 15. Logotipo de Git.

4.2.4 Azure Storage Explorer

Azure storage explorer es una herramienta para administrar fácilmente los recursos de almacenamiento con los que contamos en nuestra subscripción.

Desde esta herramienta podremos cargar, descargar y administrar archivos que tengamos en nuestro Storage Account, así como controlar y administrar también colas y tablas. (ver Figura 16).

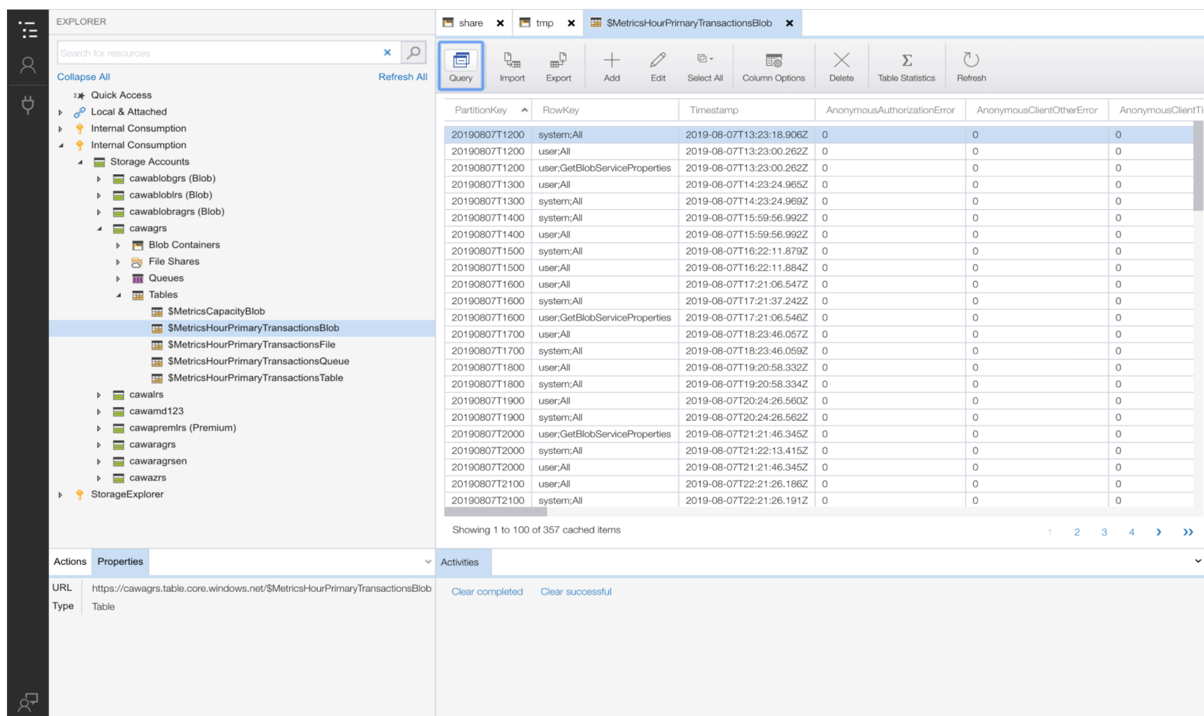


Figura 16. Interfaz Azure Storage Explorer.

4.2.5 Terraform

Terraform [25, 43] (ver Figura 17) es una herramienta diseñada para desplegar, cambiar y versionar la infraestructura de nuestra aplicación de forma segura y eficiente. Esta herramienta nos permite emplear el paradigma de la Infraestructura como Código (IaC) para nuestra aplicación.

La infraestructura como código nos permite tratar la infraestructura que hace funcionar nuestra aplicación como si fuera otro producto más. Gracias a esto, podemos replicar entornos de forma muy rápida, en lugar de tener que desplegar y configurar a mano todos y cada uno de los recursos necesarios.

Las ventajas de utilizar infraestructura como código es que la infraestructura se vuelve elástica, repetible y escalable. De esta forma, como este paradigma podemos desplegar y configurar 1 o 1000 máquinas virtuales en cuestión de minutos, disminuyendo los costes y riesgos, y aumentando la velocidad de despliegue con respecto a tener que hacerlo a mano.

Bajo este paradigma, Terraform nos permite llevar la IaC a Azure. Aunque la propia plataforma cuenta con sus propios archivos de infraestructura como código (Azure Resource Manager), Terraform nos permite hacerlo también, pero de una forma más amigable.

Los archivos de configuración de terraform describen los componentes necesarios de nuestra aplicación. Podemos incluir desde componentes de muy bajo nivel hasta componentes de alto nivel.

Además, otra ventaja de Terraform es que nos permite guardar los estados de nuestros despliegues. Con esto, cuando queramos añadir algo nuevo, no hará falta volver a desplegarlo todo a un mismo entorno, es decir, si tenemos desplegados 3 servicios y añadimos un 4, no tendremos que volver a desplegar los

4 servicios si no que Terraform se encarga de comparar el estado de nuestro entorno, mira lo que tenemos ahora mismo desplegado, con lo nuevo que queremos desplegar, y finalmente sólo efectuará el despliegue de las diferencias que salgan.

Figura 17. Logotipo de Terraform.

4.2.6 Spark

Durante los últimos años Apache Spark ha crecido enormemente, encontrando un muchísimo apoyo por parte de grandes empresas que han incentivado su desarrollo y mejora a la hora de afrontar problemas relacionados con el manejo y procesamiento de datos.

Pero ¿qué es Spark en sí?

Normalmente cuando pensamos en un ordenador, pensamos en nuestro ordenador de sobremesa o portátil, y en el uso típico que le damos, para ver películas, jugar, realizar documentos de ofimática o desarrollar páginas webs o pequeños proyectos. Sin embargo, todos hemos experimentado en algún momento como si nuestro ordenador no fuera lo suficientemente potente para satisfacer las necesidades de algún programa o proceso en nuestro ordenador.

Un área muy común donde suele ocurrir esto es en el procesamiento de datos. Los ordenadores ordinarios de casa no cuentan con la suficiente potencia para poder realizar millones de cálculos sobre millones de filas de datos. Es aquí donde entran en escena los *clusters*.

Un cluster es un grupo o conjunto de ordenadores que comparten recursos entre ellos consiguiendo de esta forma acumular la potencia de cómputo de multitud de ordenadores como si fuera uno solo. Pero esto no es suficiente. Un grupo de ordenadores está muy bien, pero es necesario disponer de un framework que pueda distribuir la carga de trabajo entre los diferentes nodos de nuestro cluster.

Es esto precisamente lo que hace Spark, controla y coordina la ejecución de tareas a través de los diferentes ordenadores.

Para organizar todas estas tareas, Spark utilizará uno de los *clusters* como *driver*. Este *cluster* será el encargado de mandar y recibir el trabajo a cada uno de los otros *clusters*. Mientras que el *cluster* que dirige se llama driver, los *clusters* que realizarán la mayor carga de trabajo se llamarán *executors*.

Nuestro *driver* se encargará de ejecutar la función main(), que estará localizado en el nodo de un cluster. Este será responsable de 3 tareas:

- Mantener la información sobre nuestra aplicación de Spark.
- Responder a nuestro programa o a una orden de entrada.
- Distribuir y planificar las tareas a realizar a través de los *executors*.

Como podemos ver, el *driver* es el corazón de Spark, es lo que mantiene viva nuestra aplicación y permite que todo funcione correctamente. De esta forma, los *executors* se encargarán de ejecutar la tarea asignada por el *driver* y de informar en todo momento del estado de la tarea que está realizando, de forma que cuando acabe, devolverá al *driver* el informe de cómo ha terminado su ejecución. Además, como podemos ver en la Figura 18, el *cluster manager*, es el encargado de dar los recursos necesarios a la Aplicación de Spark. (ver Figura 18).

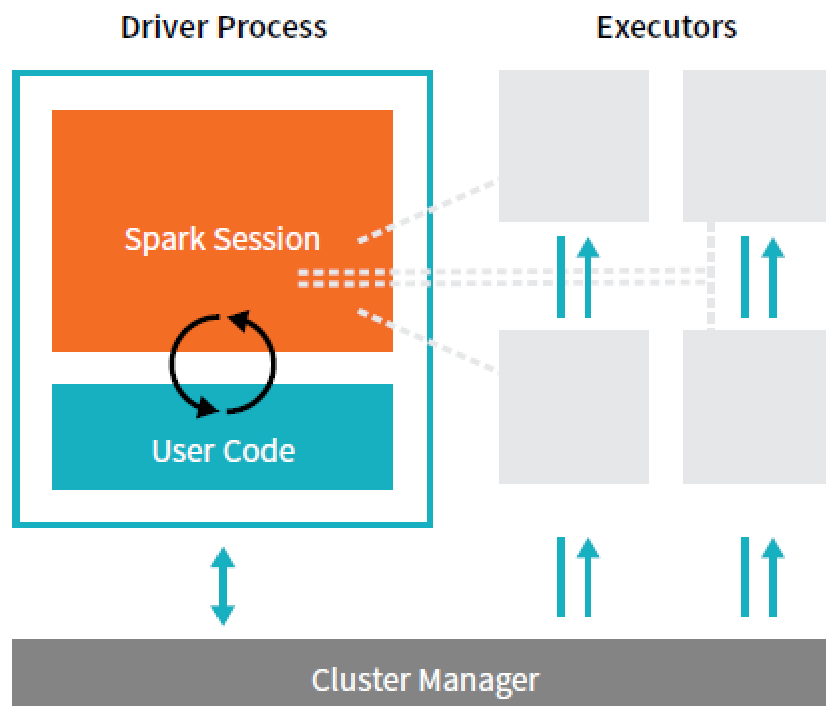


Figura 18. Flujo de trabajo en Spark. [39]

Otro aspecto a tener en cuenta es que mientras que los *executors* estarán ejecutando código de Spark siempre, nuestro *driver* puede ejecutar código de múltiples lenguajes de programación a través de la API de Spark.

Entre los lenguajes disponibles están:

- Scala
- Java
- Python

- SQL
- R

En cualquiera de los lenguajes de programación anteriores, nuestra aplicación funcionará de la misma manera, manteniendo los mismos conceptos principales. Cuando ejecutemos una aplicación de Spark, tendremos que declarar una sesión de Spark (*SparkSession*) que será el punto de entrada de nuestra aplicación para ejecutar el código de Spark. Nuestra *SparkSession* hará uso de un contexto de Spark (*SparkContext)* para poder acceder a todas las funcionalidades de Spark y de esta forma permitir al *driver* controlar la comunicación entre los *clusters* y enviar a cada uno las tareas que tienen que completar. (ver Figura 19).

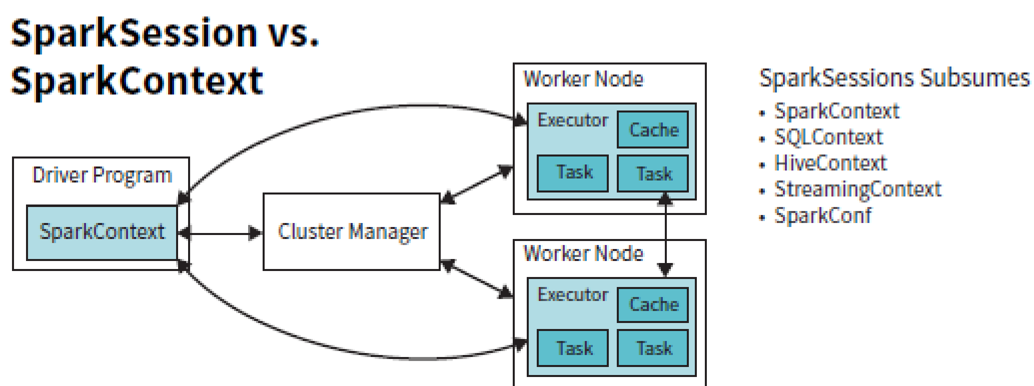


Figura 19. Interacción entre componentes en Spark. [40]

Cuando ejecutamos código en Python por ejemplo, el propio lenguaje se encargará de traducir el código en Python a código que Spark pueda ejecutar en los *executors*.

Cuando trabajemos con Spark, vamos a trabajar normalmente con *DataFrames*. Un *DataFrame* es una estructura de datos que representa una tabla de datos con filas y columnas. Un *dataframe* tiene un esquema que define de qué tipo son cada una de las columnas que contiene. Cuando tenemos un *dataframe* en spark, esta tabla estará guardada a lo largo de múltiples ordenadores. (ver Figura 20).

Time (Str)	Site (Str)	Req (Int)	Time (Str)	Site (Str)	Req (Int)	Time (Str)	Site (Str)	Req (Int)	Time (Str)	Site (Str)	Req (Int)
ts	m	1304	ts	d	3901	ts	m	1172	ts	m	2538
ts	d	2237	ts	d	2491	ts	m	2137	ts	d	2837
ts	m	1600	ts	d	2288	ts	d	3176	ts	d	3400
Partition 1			Partition 2			Partition 3			Partition 4		

Figura 20. Estructura de un *dataframe*. [40]

El hecho de tener nuestro *dataframe* divido a lo largo de múltiples ordenadores es para dividir la carga de trabajo y no tener que almacenar todos los datos en un simple ordenador.

Con respecto a esto, habrá ocasiones donde tengamos que particionar nuestro *dataframe* para dividir aun más la carga de trabajo a los largo y poder continuar con las múltiples tareas en paralelo. Una partición es un conjunto de filas que están todas situadas en un mismo ordenador. Las particiones de un dataframe nos pueden mostrar como están divididas físicamente nuestros datos a lo largo de los diferentes ordenadores.

Otro punto importante a conocer cuando trabajemos con Spark y *dataframes* es que no podremos modificar nuestros *dataframes*. Por lo tanto, cuando queremos realizar transformaciones y cambios a nuestro *dataframe*, lo que ocurre es que se generará un nuevo *dataframe* con el resultado de la transformación.

También hemos de tener en cuenta que tipo de transformación estamos llevando a cabo. Cuando hacemos transformaciones que son simples, como por ejemplo una transformación con un “*where*” para filtrar datos, tendremos una transformación que afectará solo a una partición. (ver Figura 21)

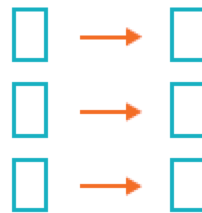


Figura 21. Transformación simple en Spark.[39]

Sin embargo, cuando realizamos transformaciones más complejas que requieren que una partición tenga que mirar datos que se encuentren en otra partición, es posible que nos encontremos con problemas de optimización o que Spark necesite escribir datos en disco de otras particiones para poder realizar cálculos o comparaciones. Esto es lo que se conoce como *shuffle*. Este es uno de los temas más importante a la hora de trabajar con Spark. En la figura 22 se puede ver el otro tipo de transformación.

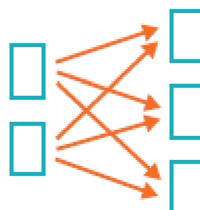


Figura 22. Transformación compleja en Spark.. [39]

Es importante conocer los dos tipos de transformación que existen en Spark para poder alcanzar altos niveles de optimización y en casos de que el rendimiento baje demasiado o tengamos altos niveles de *Shuffle* poder saber a qué se debe.

Ahora que hemos visto cómo funcionan las transformaciones en Spark, podemos hablar de uno de los conceptos más característicos de Spark, "Evaluación perezosa" o como se denomina en inglés "*lazy evaluation*".

Este concepto se basa en que cuando queramos hacer transformaciones y operaciones en un *dataframe,* Spark no ejecutará lo que queremos de forma inmediata. Lo que hará será analizar las operaciones que queremos realizar para sobreescribir el plan de ejecución actual y ejecutar un plan de ejecución optimizado para consumir la menor cantidad de recursos posible y terminar las tareas que queremos hacer en la menor cantidad de tiempo posible.
A parte de los *dataframes* Spark cuenta con varias estructuras de datos más. Cuenta con *Datasets, sql tables y Resilient Distributed Datasets (RDDs).* Estas estructuras de datos representan también datos de forma distribuida, aunque utilizan diferentes interfaces para funcionar. Sin embargo, la estructura más fácil de usar y más eficiente son los *dataframes.*

4.2.7 Structured Streaming

A pesar de que Spark tiene una corta vida, su velocidad de desarrollo e implementación de nuevas funcionalidades evoluciona muy rápido. A día de hoy son muchas las empresas que van en busca de tener sistemas capaces de procesar datos en tiempo real o que puedan dar el salto fácilmente a este ámbito.

Esto es lo que se conoce como modelos de programación en streaming (*streaming programming model*). Con esto se consigue apoyar sistema *end-to-end* que son capaces de procesar datos en tiempo real. Las aplicaciones que son capaces de reaccionar a datos de entrada en tiempo real son llamadas "aplicaciones continuas" (*continuous applications*).

Es aquí donde entra en acción *Structured Streaming*. Esta herramienta es una API de alto nivel de Spark que nos permite procesar entradas de datos tanto en *batch* y en *streaming*. *Structured Streaming* trata un *stream* de datos como si fuera una tabla pero sin límites, es decir, no tiene final la tabla (ver Figura 23). De esta forma cada vez que llegan nuevos datos, estos se añaden a la tabla.

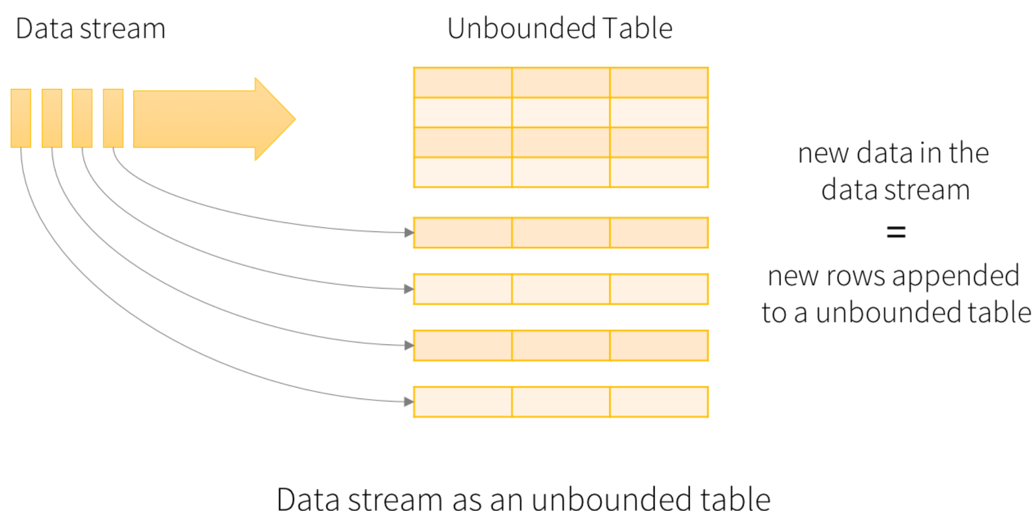


Figura 23. Structured Streaming ingestion.[23]

Structured Streaming nos permite controlar de una forma muy sencilla a partir de qué fecha queremos estar recibiendo datos. De esta forma, podemos definir lo que se llama un *Watermark* para filtrar los datos que nos llegan, procesando por ejemplo solos los datos que no lleguen y no tengan una antigûedad superior a 1 hora o 15 minutos.

La forma en que esto funciona es que, dándole nosotros a Spark una ventana de tiempo, 10 minutos, por ejemplo, el motor de Spark será capaz de filtrar estas filas automáticamente. Gracias a esto, podremos controlar el tamaño de entrada de nuestros datos, evitando que nos quedemos sin memoria en nuestros *clusters* o el hecho de poder utilizar *clusters* más pequeños y tener menores costes. Esto es porque esta herramienta para funcionar cuenta con varias partes a la hora de procesar los datos.

Cuando nosotros comenzamos a procesar un conjunto de datos en streaming, la primera tabla que se nos generará es la tabla de entrada. Esta tabla comenzará a recibir datos e irá creciendo como hemos visto antes. Gracias al *watermark* podemos "limitar" el tamaño de nuestra tabla.

Una vez tenemos los datos en nuestra tabla de entrada, no siempre será el caso en multitud de ocasiones querremos hacer transformaciones a nuestros datos. Muchas de estas transformaciones necesitan guardan el estado de los cambios en otra tabla. Este estado sería la segunda tabla que intervendría en el proceso.

Una vez realizadas las transformaciones a nuestros datos, se guardarán los resultados dentro de una tabla de resultados. En esta tabla es donde guardaremos las filas que podrían ser candidatas a ser escritas en algún sitio, las filas que queremos guardar. La tabla de resultados sería nuestro tercer actor en el proceso.

Por último, entraría en acción la última tabla, la tabla de salida. En esta tabla es donde se guardarán las filas que han cambiado o llegado nuevas y que son las que vamos a necesitar guardar o escribir. En el diagrama de la figura 24 podemos ver cómo sería el proceso completo que acabamos de ver.

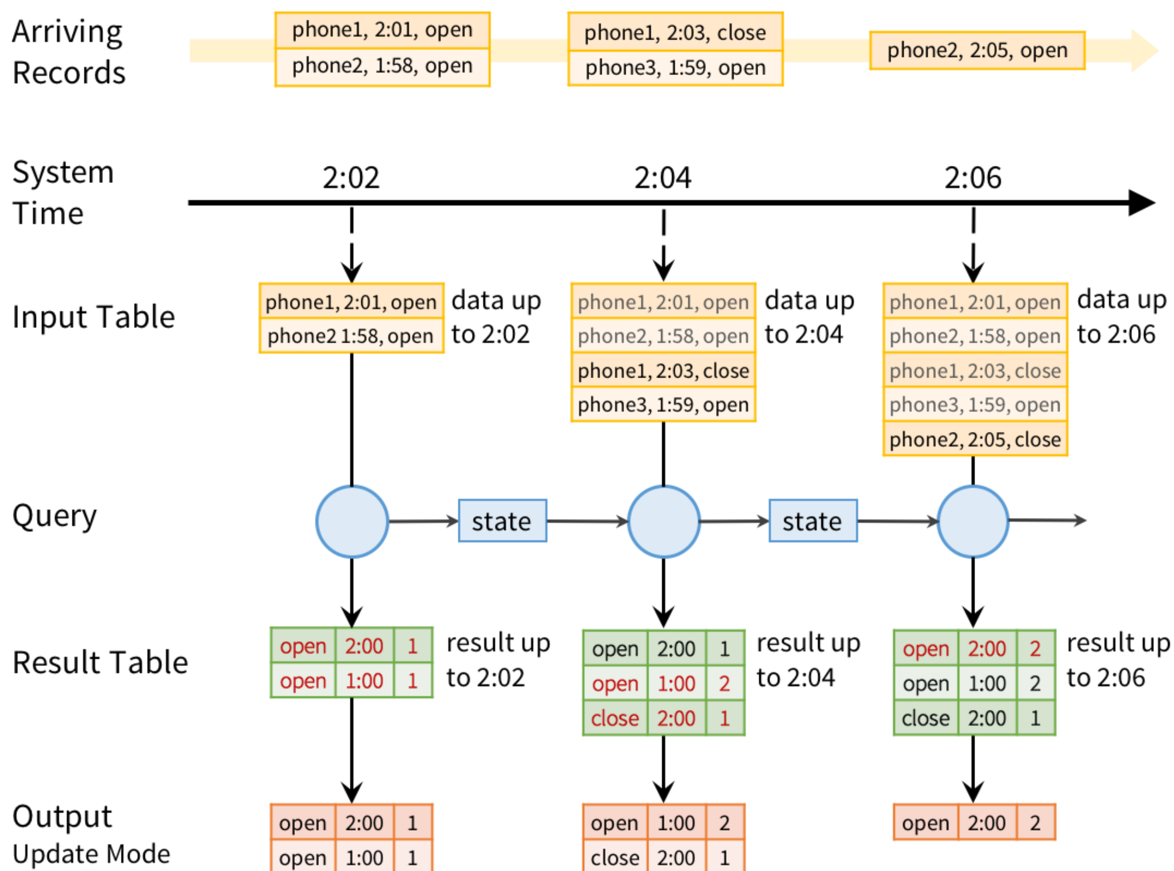


Figura 24. Diagrama de trabajo. Structured Streaming. [23]

Uno de los problemas más típicos a la hora de trabajar con *Structured Streaming* es el hecho de lidiar con el estado. En muchas operaciones nuestro estado puede ser que crezca demasiado llegando a ocupar gran parte de la memoria. Es por esto que es muy importante controlar la cantidad de datos que procesamos al trabajar con esta herramienta para evitar fallos de rendimiento debido a fallos de memoria en nuestros *clusters*.

4.2.8 Power BI

Power BI (ver Figura 25) se trata de una herramienta basada en un conjunto de servicios, aplicaciones y conectores que trabajan unidos para transformar entradas de datos en información relevante, de forma interactiva y atractiva a la vista.

Power BI puede recibir datos desde un simple fichero como bien puede ser un JSON o un CSV hasta datos que provienen de servicios de *streaming* o bases de datos.

La función principal de esta herramienta es generar informes para uso personal o empresarial con el objetivo de exponer datos de una forma que ayude a la toma de decisiones y a entender qué quieren decirnos los datos que hemos procesado.

El flujo de trabajo con Power BI comienza por enlazar nuestro entorno de desarrollo con un origen de datos. Una vez vinculados, podremos realizar transformaciones o agregar nuevas medidas a partir de los datos recibidos del origen. Tras esto, podremos construir nuestro reporte de datos, con diferentes tipos de gráficas que van desde diagramas de sectores hasta mapas de calor interactivos.

Figura 25. Logotipo de Power BI.

4.2.9 Delta Lake

Delta Lake (ver Figura 26) es una de las herramientas más novedosas y actuales ahora mismo en el mundo del *Big Data*. Se trata de un software de código abierto que es utilizado para crear capas de almacenamiento sobre los *Data lakes*.

Figura 26. Logotipo de Delta lake.

Para almacenar los datos hace uso del formato *Apache Parquet* [26]. Parquet es un formato de almacenamiento de datos basado en columnas (ver Figura 27). A diferencia de otros formatos de almacenamiento basados en filas como puede ser un csv, Parquet es capaz de almacenar los datos en forma de columna. Pero este almacenamiento no lo hace de la forma convencional, ya que no guarda en sí todos los datos de cada columna para cada fila. Lo que hace Parquet es guardar en primer lugar los diferentes valores que puede tener una columna y genera una referencia a cada uno de estos valores. Después lo que hace es construir la estructura que tendría nuestros datos, pero en lugar de asignarle a cada elemento de una columna su valor, lo que hace es asignar la referencia a su valor. De esta forma, Parquet genera *chunks* (fragmentos de información) y los codifica de forma que luego le sea más fácil de leer así como ahorra espacio de almacenamiento.

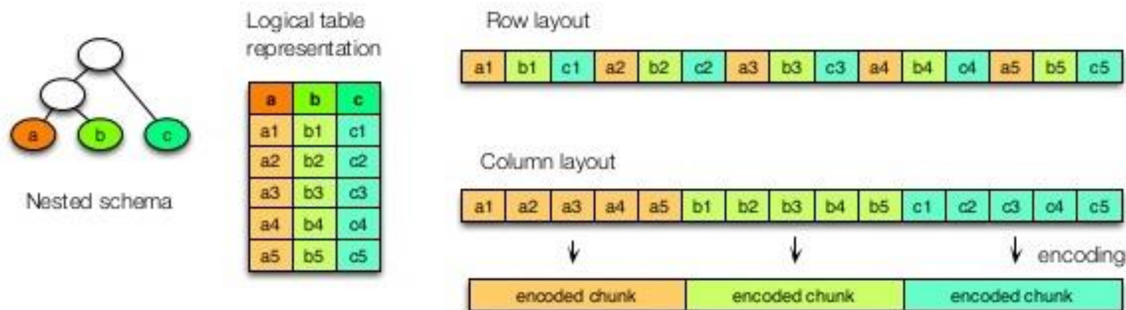


Figura 27. Diagrama sobre como Parquet almacena los datos.[41]

El hecho de que Delta Lake use Parquet para guardar los datos es que este formato de archivos permite hacer un seguimiento de cómo nuestros datos han cambiado a lo largo del tiempo. Podemos tener índices y estadísticas sobre nuestros archivos con el objetivo de incrementar la eficiencia de las transacciones que hagamos con nuestros datos.

Este nuevo formato de archivo también nos permite realizar transacciones *ACID* (Atomicidad, consistencia, aislamiento y durabilidad), como si de una tabla de una base de datos se tratase.

De hecho, es muy común llamar a un Delta Lake como si fuera una Tabla de Delta. Estas tablas cuentan además con un log. Este log guarda los cambios que se le han hecho a la tabla durante el tiempo.

Como hemos dicho antes, *Delta Lake* permite realizar operaciones *INSERT* y *UPDATE* garantizando las propiedades ACID. Una de las propiedades ACID es la atomicidad. Esto quiere decir, que si ocurre un fallo durante el proceso de una de estas acciones en nuestros archivos, los datos no se quedarán corruptos o parcialmente escritos.

Esto se consigue a través del ya mencionado Log de transacciones. Si algo no está registrado en este log, entonces es que no ha ocurrido.

El Log de transacciones funciona de tal manera que, cuando un usuario ejecuta una operacion de *insert, update o delete* Delta Lake automáticamente separa está acción en una serie de pequeños pasos compuestos de 1 o más acciones.

Este Log de transacciones se escribe automaticamente a una carpeta dentro de nuestro Delta cuando se crea por primera vez. (ver Figura 28). Cuando se hacen cambios a la tabla de Delta, estos cambios se reflejan en el Log y son guardados como commit individuales. Cada commit es un conjunto de cambios que es escrito a un archivo JSON (JavaScript Object Notation).

Figura 28. Log de transacciones de un Delta. [22]

Cuando se realizan nuevos cambios a nuestra tabla, el log de transacciones será actualizado automáticamente con las acciones que han tenido lugar generandose un nuevo archivo JSON que describirá los cambios realizados (ver Figura 29).

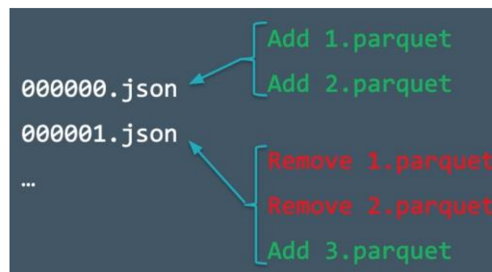
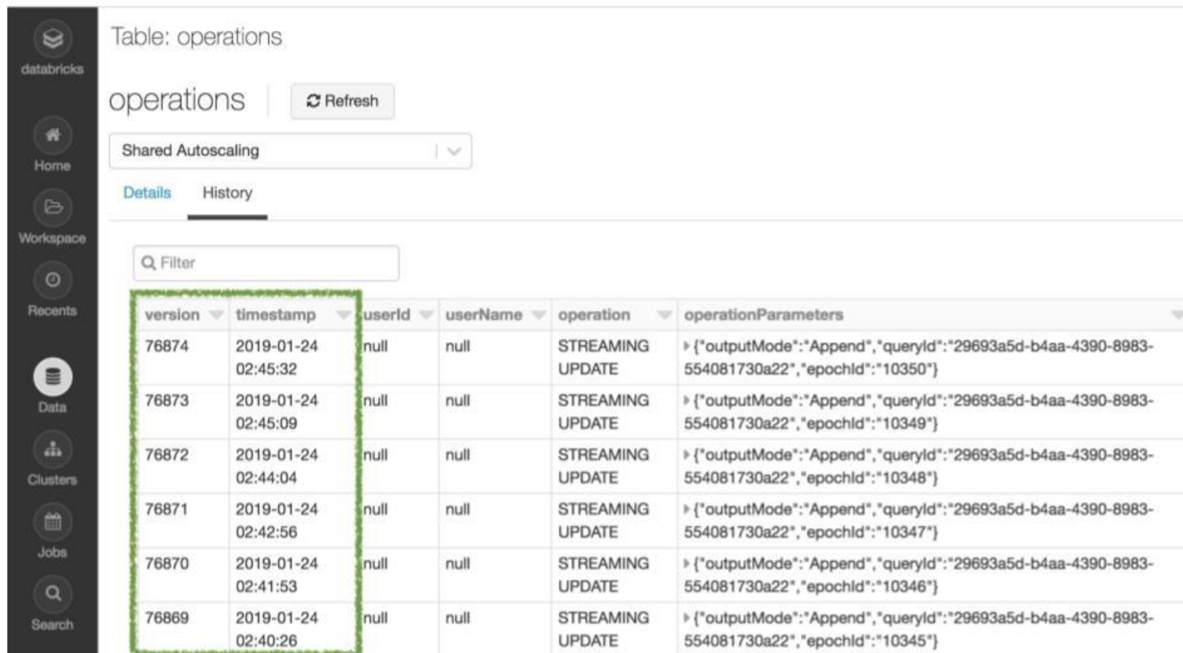


Figura 29. Delta Table. Cambios contenidos en archivos del log de transacciones. [22]

Una de las ventajas que nos otorga las tablas de Delta también es que podemos hacer "viajes en el tiempo" a través de nuestros datos y ver cuál era el estado de nuestros datos 7 versiones atrás por ejemplo. Por así decirlo, Delta Lake nos permite tener un sistema de archivos super eficiente y eficaz manteniendo las distintas versiones de los archivos como si de un repositorio se tratase.

Esto es lo que se conoce como *"Time travel"* o *"Data Versioning"*. Cada tabla es la suma total de todos los commits que se han guardado en el log de transacciones de la tabla de Delta (ver Figura 30). Como en el log tenemos un registro paso a paso de todos los cambios que han tenido lugar en nuestra, entonces podemos recrear a la inversa todos y cada uno de los cambios que han tenido lugar, pudiendo volver de esta manera a versiones anteriores de nuestros datos.



version	timestamp	userId	userName	operation	operationParameters
76874	2019-01-24 02:45:32	null	null	STREAMING UPDATE	{"outputMode":"Append","queryId":"29693a5d-b4aa-4390-8983-554081730a22","epochId":"10350"}
76873	2019-01-24 02:45:09	null	null	STREAMING UPDATE	{"outputMode":"Append","queryId":"29693a5d-b4aa-4390-8983-554081730a22","epochId":"10349"}
76872	2019-01-24 02:44:04	null	null	STREAMING UPDATE	{"outputMode":"Append","queryId":"29693a5d-b4aa-4390-8983-554081730a22","epochId":"10348"}
76871	2019-01-24 02:42:56	null	null	STREAMING UPDATE	{"outputMode":"Append","queryId":"29693a5d-b4aa-4390-8983-554081730a22","epochId":"10347"}
76870	2019-01-24 02:41:53	null	null	STREAMING UPDATE	{"outputMode":"Append","queryId":"29693a5d-b4aa-4390-8983-554081730a22","epochId":"10346"}
76869	2019-01-24 02:40:26	null	null	STREAMING UPDATE	{"outputMode":"Append","queryId":"29693a5d-b4aa-4390-8983-554081730a22","epochId":"10345"}

Figura 30. Ejemplo de tabla de versiones con los últimos commits realizados. [22]

Entre todas las ventajas (ver Figura 31) de Delta Lake podemos encontrar:

- Unificación del procesamiento en *batch* y en *streaming*.
- Es capaz de lidiar con cambios en los esquemas de los archivos automáticamente u obligar a que se fuerce un esquema.
- *Time travel*: Versionado de los datos de nuestra tabla de delta para poder volver a una versión anterior en cualquier momento, así como auditar fallos que hayan podido ocurrir o realizar experimentos con machine learning.
- Operaciones *Upsert* y *Delete*: Las tablas de delta soportan operaciones de merge para añadir datos a una de nuestras tablas de Delta de forma que sólo añadiremos lo nuevo, actualizaremos datos ya existentes o eliminaremos filas en base a condiciones.
- Posibilidad de realizar transacciones ACID con nuestros archivos.

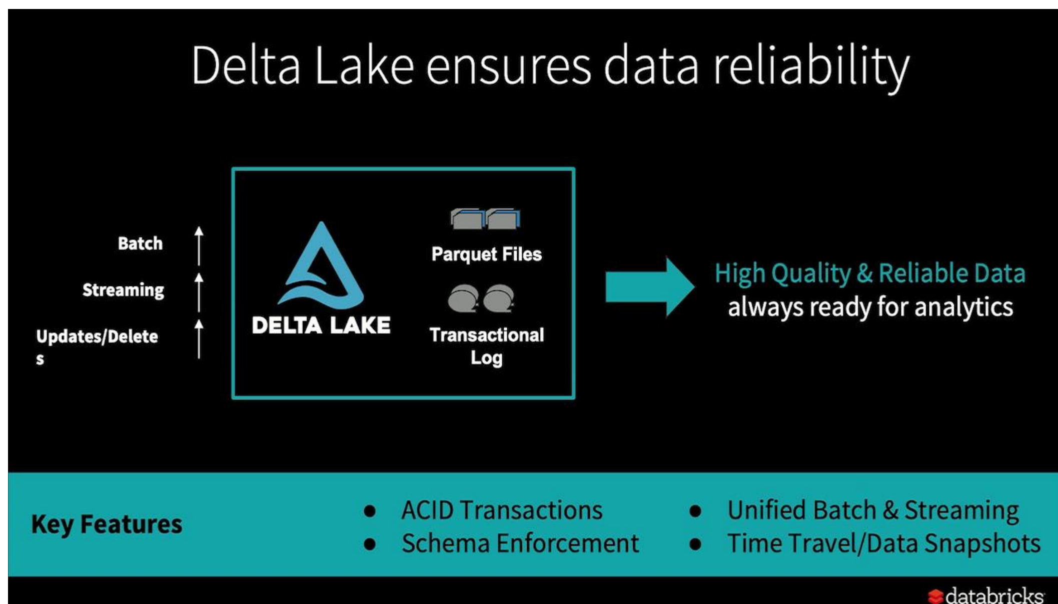


Figura 31. Características clave de Delta Lake. [22]

4.3 Servicios

En este apartado nos vamos a centrar solamente en los servicios de Azure que se han usado para desarrollar el proyecto.

4.3.1 Azure DevOps

Azure DevOps [27] (ver Figura 32) es un servicio de Azure usado por equipo de desarrollo y soporte con el objetivo de planificar el trabajo a desarrollar, como repositorio para los archivos de código fuente de nuestros sistemas y para construir y desplegar nuestras aplicaciones.

Azure DevOps está compuesto internamente por diferentes servicios. Cada uno de estos servicios integra diferentes características que ayudan con el desarrollo y planificación de una aplicación. Estos servicios son:

- **Azure Repos**: Es como GitHub. Proporciona repositorios de Git para poder tener un control de versiones de nuestro código.

- **Azure Pipelines**: Proporciona un entorno para construir nuestra aplicación y prepararla para ser desplegada. Con este servicio podemos preparar un sistema que reaccione por ejemplo a Pull Request o Commits en una rama de Azure Repos o en cualquier otro sistema de control de versiones como puede ser GitHub (Siempre y cuando el sistema externo cuente con integración con Azure DevOps) para construir o realizar unas acciones y así poder automatizar el despliegue de nuestras aplicaciones.

- **Azure Boards**: Proporciona todo un conjunto de herramientas para ayudarnos a la hora de llevar a cabo una metodología Ágil. Con este servicio podemos llevar a cabo un seguimiento

del trabajo realizado, así como controlar errores o posibles incidentes mediante el uso de tableros Kanban y otras herramientas que apoyan los métodos de Scrum.

- **Azure Test Plan:** Proporciona un conjunto de herramientas para testear nuestras aplicaciones. Aquí podemos declarar y preparar un conjunto de test para que sean lanzados con cada versión de nuestra aplicación. Estos test también podríamos lanzarlos desde Azure Pipelines.

- **Azure Artifacts:** Este servicio permite a los equipos de desarrollo a compartir librerías o proyectos en Maven, npm o NuGet para ser usados pública o privadamente por un equipo.

Figura 32. Logotipo de Azure DevOps.

4.3.2 Azure Data Factory

Azure Data Factory se trata de un servicio de Azure que nos permite orquestar los procesos de extraer, transformar y cargar (Load, Transform, Load - ETL) típicos de los procesos de Big Data.

Con este servicio se es posible de descargar datos de API, mover grandes cantidades de datos de un lugar a otro, así como llamar a otros servicios de Azure para ejecutar otra parte de la lógica necesaria que necesite el proceso como puede ser ejecutar un notebook de Azure Databricks, un fragmento de código en una Azure Function o encender o apagar un Data Warehouse.

Para trabajar con Azure Data Factory tendremos que definir una actividad para construir nuestro Pipeline. En función de lo que necesitemos hacer, nuestro Pipeline contará con más o menos actividades. Un Pipeline puede luego contener otros Pipelines en su interior para separar la lógica y mantener la legibilidad. Nuestro Pipeline será llamado por un trigger que definiremos para que se ejecute en un momento dado. Este momento puede ser a una hora determinada, cada X tiempo o al reaccionar a un evento.

Dentro de este servicio también contaremos con unos elementos llamados Linked Services. A través de estos elementos podremos vincular nuestro Data Factory con otros servicios, externos e internos a Azure.

Azure Data Factory también nos permite conectarnos de forma segura a servidores privados que estén dentro de su propia red privada a través de un *Integration Runtime*.

4.3.3 Azure SQL Database

El servicio de Azure SQL Database [29] es una plataforma como servicio que nos ofrece un motor de bases de datos bajo demanda. Con este servicio se pueden crear bases de datos totalmente administradas con nosotros sin la necesidad de tener que preocuparnos por el servidor.

Con Azure SQL Database podemos contar con una base de datos que se apagará cuando no tengamos ningún servicio y a ninguna persona usándola. Podemos servir los datos desde este servicio para poder utilizar estos datos en informes en Power BI

4.3.4 Azure Databricks

Azure Databricks (ver Figura 33) es una plataforma como servicio que integra y acerca el uso de Spark al usuario a través de múltiples lenguajes de programación. Databricks pone a nuestra disposición la posibilidad de configurar todo un entorno de *clusters* controlados por Spark para construir arquitecturas de procesamiento de datos y análisis con Machine Learning y Deep Learning.

Databricks soporta procesamiento de datos tanto en *Batch* como en *Streaming*. Podemos crear entornos de producción seguro y confiables en Azure, definiendo una serie de *clusters* que en caso de necesitar más potencia de procesamiento, podrán escalarse automáticamente para abastecer la potencia necesaria. Esta elasticidad es dinámica. Esto quiere decir que una vez que no necesitamos esa potencia los clústeres se disminuyen automáticamente de nuevo hasta la configuración mínima que nosotros habíamos especificado.

Además, contamos con distintas versiones de Spark, accesibles rápida y fácilmente a través de la configuración de los *clusters*.

Otro punto a favor de este servicio es que permite el trabajo colaborativo, pudiendo estar varias personas trabajando sobre el mismo notebook o diferentes notebooks al mismo tiempo.

Por otro lado, al igual que sucede con otros servicios, Databricks es capaz de conectarse con otros servicios. De esta forma, por ejemplo, Databricks puede conectarse con el servicio de Azure Key Vault para obtener claves y contraseñas u otro tipo de datos sensibles que no queramos tener en nuestros notebooks. De esta forma podemos aumentar la seguridad de nuestro sistema, no exponiendo ninguna contraseña en nuestro código. De igual modo, Databricks cuenta con más conectores para otros servicios, como puede ser la conexión a *Storage accounts* para leer y escribir datos o la conexión a bases de datos como Azure SQL Database o Azure Synapse Analytics.

Figura 33. Logotipo de Databricks.

4.3.5 Azure Function App

Las Azure Functions (ver Figura 34) son uno de los servicios más usado dentro de Azure por su naturaleza. Se trata de un servicio que nos permite ejecutar fragmentos de código, funciones para realizar tareas muy específicas o tareas muy sencillas o complejas sin demasiada carga de trabajo sin necesidad de contar con un servidor ni ninguna infraestructura que ejecute nuestro código. Azure se encarga en todo momento de proporcionar los recursos necesarios para ejecutar nuestras funciones.

Para que una función se ejecute tendremos que llamarla. Esto significa que tendremos que dar un evento para que la función se lance. Estos eventos pueden ser de diferente índole. Podemos tener *triggers* que lancen la Azure Function al llamarla por un método HTTP (*Hypertext Transfer Protocol*) como podría ser un método *GET* o un método *POST*. Puede ser llamada también por el transcurso de un período de tiempo, pudiendo definir que nuestra función se lance cada 5 minutos por ejemplo, o que reaccione a la llegada de un nuevo archivo a nuestro *Storage account* o aun *Storage queue*.

Todos estos eventos que hacen que nuestra Azure Function se ejecute se denominan parámetros de entrada. Una Azure Function tiene que contar mínimo con un parámetro de entrada, y puede contar con 1 o más parámetros de salida. Un parámetro de salida puede ser por ejemplo escribir un archivo a un *Storage Account*, a una cola en un *Storage Queue* o un mensaje a un *Event Hub* [30].

Dentro de nuestra Azure Function, podremos definir tantas funciones y escribir tanto código como queramos. No hay límite respecto a esto, pero sí que tenemos que tener en cuenta que el tiempo máximo de ejecución por función es de 10 minutos. Una vez se superan los 10 minutos de ejecución nuestra Azure Function dejará de ejecutarse y fallará. Para ello necesitaremos usar las *Durable Functions* [31].

Por ello es importante tener en cuenta esto, para que en caso de tener una Azure Function que tarde más de 10 minutos de ejecución, tal vez deberíamos dividir la lógica de procesamiento de nuestra función en más pasos.

Por otro lado, las Azure Functions ahora mismo cuentan con soporte para los siguientes lenguajes de programación: Python, C#, Java, JavaScript, PowerShell.

Figura 34. Logotipo Azure Functions.

4.3.6 Azure Key Vault

Azure Key Vault (ver Figura 35) es un servicio que nos ayudará con la administración de contraseñas, claves y certificados. Con este servicio podremos almacenar de forma segura las contraseñas de servicios que necesite autenticación, las claves de conexión a servicios como *Storage Accounts* o bases de datos, así como certificados.

Gracias a tener todos nuestros secretos y contraseñas centralizadas en un único lugar, podremos tener un mayor control sobre la distribución de estas, reduciendo así el riesgo de filtrar alguna clave por error.

Con este servicio ya no será necesario almacenar nunca más nuestras claves dentro de la propia aplicación para configurar conexiones. Al no tener que guardar las contraseñas como parte del código, hace que nuestras aplicaciones sean mucho más seguras.

Además, cada secreto en el Key Vault puede tener varias versiones. Para acceder a un secreto, clave o certificado, tendremos que hacerlo a través de una URI (*uniform resource identifier*). Cada URI es capaz de acceder a una versión específica de un secreto.

Por otro lado, nuestras claves no se guardan como texto plano en Azure, si no que una vez subimos nuestras contraseñas a Azure, este se encarga de cifrarlos utilizando algoritmos estándar, longitudes de clave y módulos de seguridad de hardware validados mediante estándares federales de procesamiento de información 140-2 de nivel 2. Este estándar se trata del usado por los ordenadores del gobierno de los Estados Unidos para la acreditación de módulos criptográficos.

De esta forma, con este nivel de seguridad, solo se podrá acceder al almacén de claves si se cuenta con la autorización y autenticación correctos. Esto quiere decir que a parte de contar con las claves para acceder, el servicio debe de tener autorizada el acceso de dicha persona al almacén.

Esto es así porque un Azure Key Vault cuenta diferentes actores. Por un lado tenemos al propietario del almacén de claves. El propietario del almacén de claves tiene acceso completo al recurso y control pleno sobre él. Además el propietario puede realizar auditorías para supervisar y registrar quién accede a los secretos y claves. El segundo actor en acción es el consumidor del almacén. Este actor es cualquier persona o recurso que puede realizar acciones sobre los elementos del almacén de claves. Cada actor tendrá unos permisos sobre el almacén, que serán otorgados por el propietario del almacén.

Figura 35. Logotipo Azure Key Vault.

4.3.7 Azure Stream Analytics

Azure Stream Analytics se trata de un servicio de procesamiento de eventos en tiempo real diseñado para recibir, analizar y procesar grandes volúmenes de eventos y datos desde varios orígenes.

El origen de los datos puede ser muy variado pasando desde dispositivos , sensores, etc hasta fuentes de datos provenientes de redes sociales y aplicaciones.

Un origen de datos puede utilizarse para configurar alertas, iniciar flujos de trabajo, almacenar los datos realizándose alguna transformación o proveer de información a una herramienta que disponga de un informe de datos.

Por otro lado, Stream Analytics se encuentra dentro del entorno de Azure IoT Edge [32]. Azure IoT Edge se trata de un servicio totalmente administrado por Azure que permite implementar cargas de trabajo como inteligencia artificial, servicios de Azure o de terceros o nuestra propia lógica para ser ejecutadas en dispositivos IoT Edge mediante contenedores.

La forma en que Stream Analytics funciona es que cuenta con una entrada de datos, una consulta y una salida de datos (ver Figura 36). Las entradas de datos pueden ser desde Azure Event Hubs, Azure IoT Hub o Azure Blob Storage. La consulta se basa en el lenguaje de consulta de SQL y se usar para filtrar, ordenar y unir con facilidad los datos de transmisión. Se puede extender la funcionalidad de las consultas agregando funciones definidas por el usuario (UDFs) a las consultas de SQL. Estas funciones pueden estar definidas en C# o en JavaScript. Por último, tenemos las salidas. Las salidas pueden utilizarse para enviar datos a Azure Functions, colas, etc para desencadenar otros flujos de trabajo. También se pueden enviar los datos directamente a un informe en Power BI. Por último, los datos se pueden almacenar sin más en un servicio de almacenamiento como puede ser *Azure Storage*.

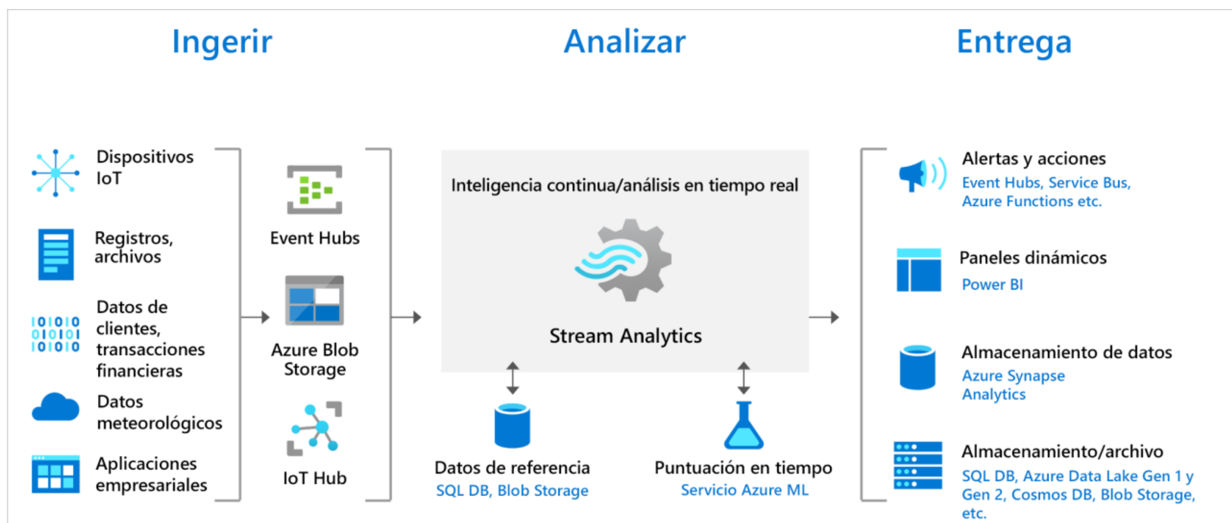


Figura 36. Flujo de trabajo de Steam Analytics. [12]

4.3.8 Azure Storage Account

Azure Storage Account se trata de un servicio de almacenamiento en la nube preparada para ser trabajada en entornos modernos.

Este servicio cuenta con varios sub-servicios internos para almacenar distintos tipos de datos (ver Figura 37). Entre ellos cuenta con contenedores para almacenar objetos de datos no estructurados. Estos datos se denominan *blobs.* Además cuenta con un sistema de recursos compartidos para archivos para implementaciones locales y en la nube. Por otro lado, cuenta también con almacenamiento en disco para ser utilizado por Máquinas Virtuales de Azure. También podemos encontrar un servicios para almacenamiento de mensajes en colas, denominado *Azure Storage Queue*. Y por último contamos con un servicio de almacenamiento en tablas llamado *Azure Storage Queue.*

Dentro de nuestro servicio de almacenamiento contamos luego también con diferentes tipos de almacenamiento. Podemos guardar nuestros datos en blob con un nivel de acceso en "*Hot"* que significa que los datos van a ser leídos y que se van a trabajar con ellos de manera cotidiana, por lo que necesitan ser accedidos rápidamente. Por otro lado, tenemos lo opuesto. Podemos establecer un nivel de acceso llamado "*Cold Storage"*. Este nivel de almacenamiento es mucho más barato y se utiliza para archivos y datos que queremos mantener pero que será muy raro que accedemos a ellos. En caso de querer acceder a ellos, tendremos que esperar un tiempo hasta poder leer el contenido.

En nuestro storage podemos definir reglas de acceso para diferentes usuarios o usuarios externos a nuestra plataforma. De esta manera, podemos definir enlaces para acceder a directorios específicos que tengan una validez de 1 día por ejemplo, o dar permiso a un usuario para acceder solo a unas carpetas dentro de nuestro almacenamiento.

Para acceder a este servicio podremos hacerlo a través de HTTP o HTTPS o en su lugar utilizar las librerías que ofrece Microsoft en diversos lenguajes como .NET, Java, Node.js, Python, PHP, etc.

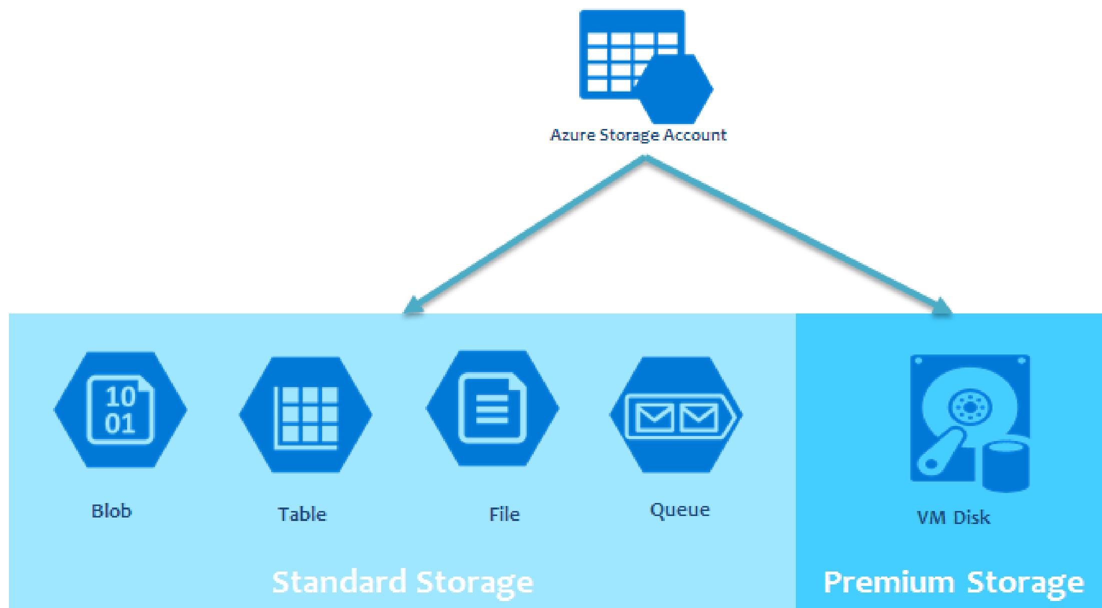


Figura 37. Servicios dentro de Azure Storage Account.[9]

4.3.9 Azure Event Grid

Azure Event Grid se trata de un servicio para reaccionar en casi tiempo real a eventos que se generan en recursos tanto de Azure como externos.

Con este servicio podemos crear sistemas basados en eventos (ver Figura 38). Azure Event Grid permite realizar filtros y configuraciones personalizadas de los eventos que se producen para reaccionar sólo a algunos de ellos de forma que sólo procesamos ciertas actividades.

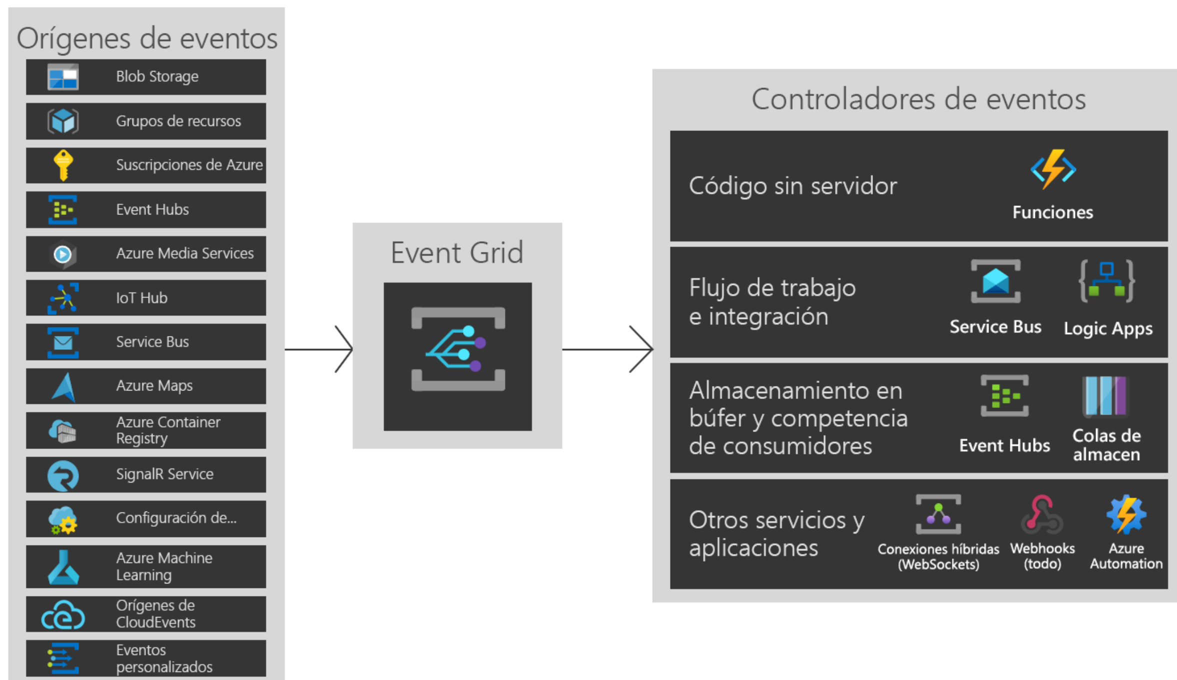


Figura 38. Flujo de trabajo con Event Grid.[15]

5. Arquitectura Delta

Vamos a pasar ahora a explicar en que se basa la arquitectura Delta [33][34] y porque ha sido esta la elegida para ser aplicada en el desarrollo de la lógica de este sistema.

5.1 ¿Qué es la arquitectura Delta?

Al igual que sucede con el desarrollo de aplicaciones web, de escritorio y móviles, el desarrollo de sistemas basado en el flujo y procesamiento de datos también se basa en unas arquitecturas y patrones de diseño.

En la actualidad, cada vez son más las empresas que necesitan extraer información relevante de los datos que producen en un menor espacio de tiempo. Con esta necesidad nace el procesamiento de datos en tiempo real. Pero ante esto, nos encontramos con un problema. Y es que el coste de procesar y extraer información de datos en tiempo real es un proceso costoso. Más aún cuando las empresas al intentar ahorrar costes durante este proceso quieren mantener otro sistema en paralelo para cuando se detenga el procesamiento en tiempo real, poder ejecutar un procesamiento por ejemplo a cada hora con todos los datos recibidos.

Es aquí donde nacen estas nuevas arquitecturas. La arquitectura Delta se trata de una arquitectura basada en la mezcla de la arquitectura Lambda utilizando la herramienta de Delta Lake junto con Structured Streaming de Spark para realizar una integración en una misma tarea de procesamiento de datos entre el procesamiento en *batch* y el procesamiento en *streaming*. (ver Figura 9).

La arquitectura Delta asume que cualquier fuente de datos que llegue a nuestro sistema será un *streaming* de forma que estos datos serán procesados de forma incremental, agregando cada vez nuevos datos a nuestras tablas de Delta. Una de las diferencias clave frente a la arquitectura Lambda es que Delta ya no considera como inmutables los datos del Data Lake, de forma que cualquier *batch* de datos que llegue a nuestro sistema y sea procesado puede actualizar los datos que ya tenemos guardados.

A la hora de construir nuestra arquitectura Delta, tendremos que tener en cuenta tanto la forma en que procesamos los datos, como la forma en la que los guardamos.

Respecto a la forma en la que estructuramos nuestro Delta Lake, podemos ver en la Figura 9 que dispondremos de diferentes capas.

En primer lugar contaremos con un nivel denominado *Landing* o capa de bronce. En esta capa es donde tendremos todos los datos de la plataforma de datos en su manera primitiva. Sin realizar ningún cambio, transformación, ni nada similar.

En una segunda capa podremos encontrarnos con el nivel conocido como *Staging* o capa de Plata. En esta capa es donde haremos las transformaciones necesarias a nuestros datos para deserializarlos y

limpiar la información que hemos recibido. En esta capa también se agregan datos, como medias, sumatorias, campos calculados, etc.

Por último, tendríamos la capa de *Curated* o capa de oro. En esta capa es donde podremos encontrarnos con los datos más importantes y destacables. Por así decirlo con los datos que tenemos en esta capa podríamos comenzar a tomar decisiones. En esta capa se encuentran los datos agregados y listos para ser exportados a bases de datos y herramientas de informes.

A la hora de procesar los datos, necesitas utilizar una herramienta o tecnología que nos permita procesar los datos de igual manera tanto para *Streaming* como para datos en *Batch.* Se suele hacer uso de la API de alto nivel de Spark llamada *Structured Streaming.*

Structured streaming nos permite lidiar con grandes volúmenes de datos y tratarlos como si fuesen todos un streaming, incluso cuando su naturaleza en sí no lo sea. De esta forma trabajaremos con nuestro stream de datos de forma normal, leyendo y transformando los datos. Cuando vayamos a escribir los datos será cuando llegue lo interesante.

Structured streaming cuenta con un par de funciones que serán clave para poder implementar esta arquitectura.

En primer lugar, contamos con un método a la hora de escribir llamada *foreachBatch()*. Este método lo que nos permitirá será realizar las operaciones sobre nuestra tabla de Delta para actualizar, eliminar o añadir nuevos datos a nuestra tabla. La forma en que funciona esta función es que separa todos los datos que necesitamos escribir en *micro-batches*. Tendremos que pasarle una función nosotros con la lógica específica que necesitamos hacer, cuyos parámetros de entrada serán como mínimo el *micro-batch* y el identificador del mismo. De esta forma cuando nuestra función reciba este *dataframe*, podemos procesarlo como si de un procesamiento en *batch* se tratara en lugar de un streaming. Aquí definir bajo qué condiciones queremos que una fila se actualice, se añada o en su defecto se elimine.

En segundo lugar, contamos con una opción llamada "*Trigger = Once*" que será la opción que nos permite ejecutar nuestro procesamiento como si de un procesamiento en *batch* se tratase. Lo que conseguimos añadiendo esta opción de configuración es que nuestro *streaming* de datos parará de procesar datos una vez termine de escribir el conjunto de datos que ha leído. En caso de querer que será continuo y que no sea una única ejecución podremos eliminar esta configuración al igual que también podemos especificar cada cuanto tiempo queremos que se escriban nuestros datos.

5.2 Otras arquitecturas para soluciones basada en flujos de datos

A parte de la arquitectura que vamos a emplear en este sistema, existen otras muchas arquitecturas. Ahora vamos a hablar de las dos más utilizadas en el mundo empresarial: la arquitectura Lambda [35] [36] y Kappa [37].

La principal diferencia entre estas dos arquitecturas es la forma en que ambas realizan el flujo de procesamiento de los datos.

La arquitectura Lambda fue definida hace varios años por Nathan Marz como propuesta para poner solución a los sistemas que las empresas necesitan empezar a construir para procesar los *Gigabytes* de datos que producen. Antes, para poder procesar esta enorme cantidad de datos en tiempo real eran necesarios utilizar algoritmos como *MapReduce* y se almacenaban los datos. El problema estaba en que cuando queríamos mostrar los datos estos tenían ya varias horas de antigûedad.

Esta arquitectura (ver Figura 39) que presentó Nathan consistía en los siguiente:

- Una capa de *Batch* que almacenaba todos los datos de entrada sin formato y realizaba el procesamiento de estos datos.
- Una capa de *Sreaming* que analizaba los datos en tiempo real teniendo una latencia baja pero sacrificando precisión.

La capa de *batch* se dividía en capas de servicio que servían de vista para poder realizar consultas eficaces y precisas.

Por la capa de *Streaming* fluyen los datos limitados por los requisitos de latencia.

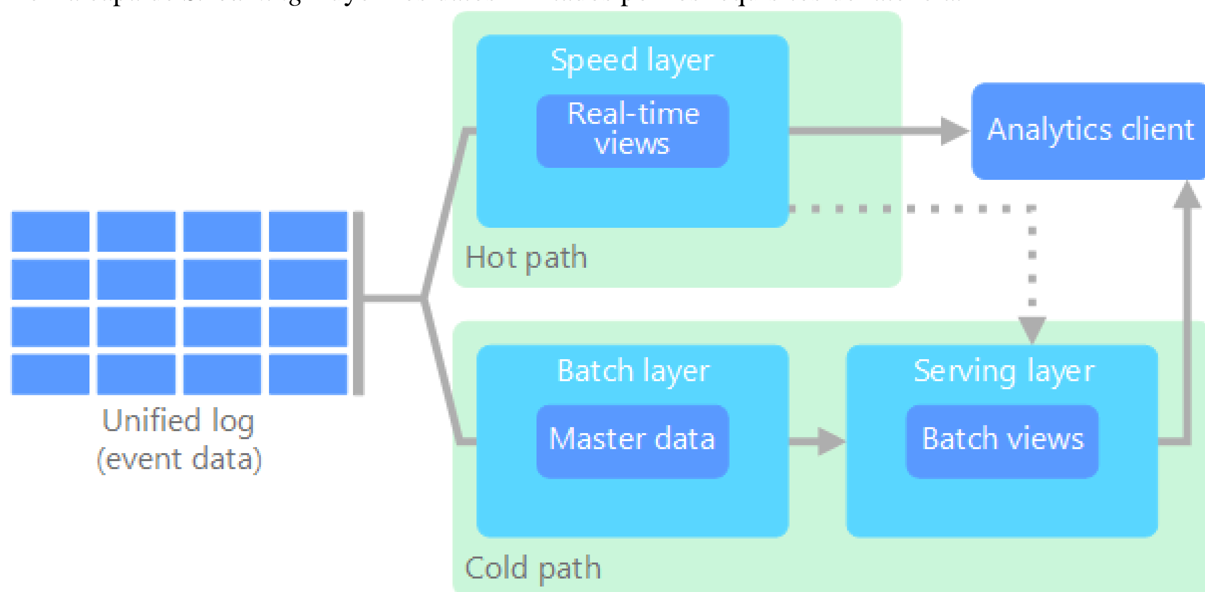


Figura 39. Arquitectura Lambda. [35]

Por otro lado, tenemos la arquitectura Kappa. Esta arquitectura fue propuesta por Jay Kreps. Aunque tiene los mismos objetivos que la arquitectura Lambda, presenta un importante cambio ya que todos los flujos de datos tienen que pasar por el mismo punto de acceso (ver Figura 40). Con esta nueva arquitectura se consiguió reducir la complejidad que tenía la arquitectura lambda al tener la lógica de procesamiento en dos sitios diferentes, por lo que también era más costosa de mantener.

En esta arquitectura todo el flujo de datos se procesa a través de de la capa de *Streaming*.

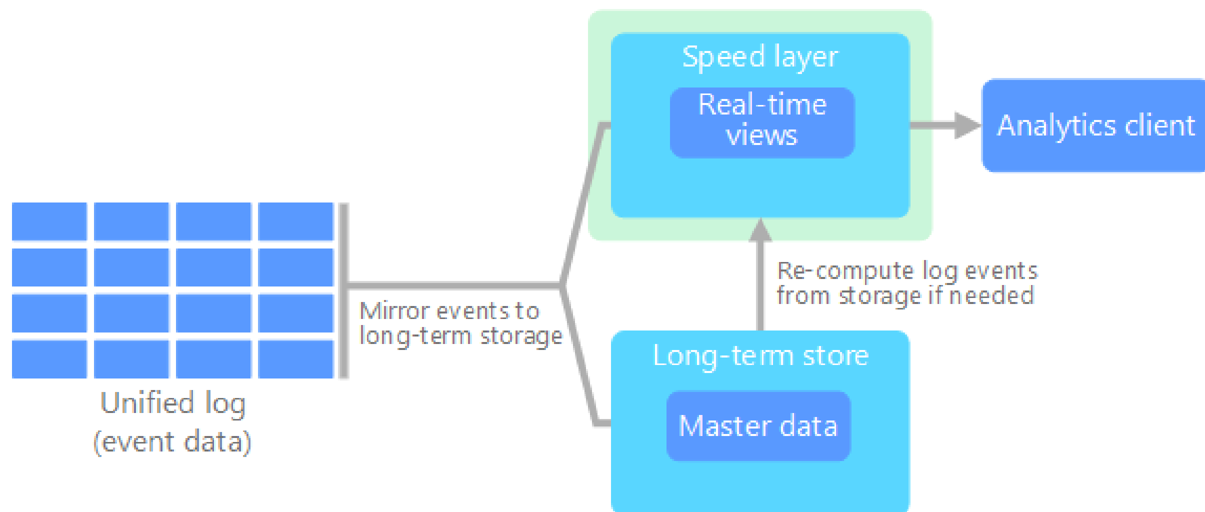


Figura 40. Arquitectura Kappa. [37]

5.3 Ventajas de la arquitectura Delta

Como hemos podido ver en el punto anterior, la arquitectura Kappa se trata de una mejora sobre la arquitectura Lambda. Pero aun así quedan más puntos que mejorar, como es la complejidad que aún presenta la arquitectura Kappa cuando se implementa en la realidad.

Los puntos débiles que tiene la arquitectura Kappa y Lambda son solucionados en la arquitectura Delta (ver Figura 41) mediante el procesamiento a través de una misma capa, pero además reduciendo la complejidad de este proceso mediante el uso de Structured Streaming y Delta Lake, haciendo que los pipelines desarrollados mediante esta arquitectura sean mucho más mantenibles.

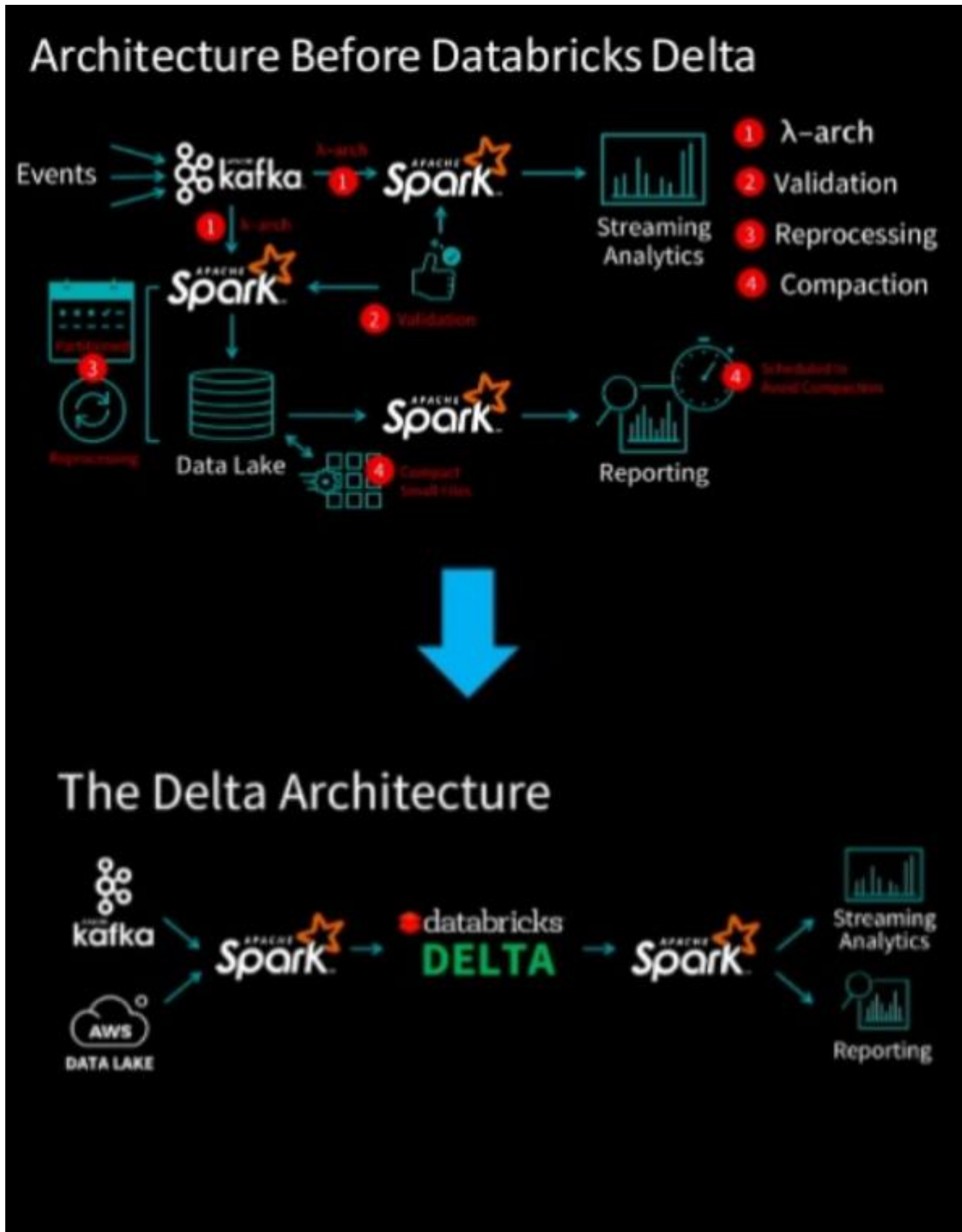


Figura 41. Comparación entre arquitectura Lambda o Kappa vs Delta. [22]

Uno de los problemas que encontramos en las otras arquitecturas era la posibilidad de poder saber a ciencia cierta qué datos estamos leyendo mientras estos mismos estaban siendo escritos. Esto quiere decir, que si dos actores leían la misma carpeta mientras otro estaba escribiendo, era muy posible que

los datos leídos por ambos actores fueran diferentes porque no había forma de garantizar que ambos iban a leer exactamente lo mismo.

Esto es algo que soluciona la arquitectura Delta, gracias al uso de Delta Lake. Esto logramos solucionarlo gracias a una de las características de Delta que mantiene un entorno aislado entre los actores que pretenden escribir datos y los actores que pretenden leer esos datos.

De esta forma, cuando haya un actor escribiendo, y dos actores leyendo, los actores que están leyendo los datos van a leer exactamente los mismos datos.

Otro problema que soluciona la Arquitectura Delta es que cuando queríamos leer de forma incremental un conjunto de datos muy grande, no es posible mantener un buen rendimiento. Esto también lo conseguimos con Delta Lake gracias a la forma en la que almacena los datos. Delta Lake cuenta con un sistema de archivos optimizado apoyado de un sistema de metadatos que ayuda a manejar estos archivos. Gracias a los checkpoints que guarda Delta, podemos mantener un rendimiento óptimo.

Un tercer problema que conseguimos solucionar es la posibilidad de volver a una versión anterior de los datos en caso de que ocurra algún fallo durante el procesamiento de los archivos o mientras se están escribiendo. Esto lo conseguimos gracias a la característica de Delta llamada "*Time Travel*" donde a consecuencia de los archivos del log de transacciones sabemos en todo momento qué es lo que ha ocurrido en nuestro Delta y de esta forma podemos recuperar versiones antiguas de nuestros datos.

Un cuarto problema que encontrábamos era que no si había datos que habían llegado tarde no los habíamos procesado en el momento, no teníamos forma de procesar estos archivos o datos de nuevo si no era procesando todo de nuevo. Esto lo solucionamos con esta arquitectura gracias a *Structured Streaming* y Delta, ya que en los checkpoints que guardamos sabemos en todo momento qué datos han sido los que se han procesado y cuáles no, de forma que si hay datos nuevos pero que deberían haberse procesado hace un tiempo, podremos procesarlos sin ningún problema sin tener que hacer nada especial ni agregar ninguna lógica a nuestro procesamiento.

Por último, antes no teníamos la posibilidad de replicar el procesamiento de datos históricos con los nuevos datos que llegaban nuevos. Gracias a Delta somos capaces de solucionar este problema.

Una de las recomendaciones al utilizar esta arquitectura es realizar siempre que se pueda un streaming de datos de tabla de Delta a tabla de Delta para disminuir más aún la latencia de los datos.

A parte de todos estos problemas que solventamos utilizando la arquitectura Delta, también contamos con las siguientes ventajas (ver Figura 42):

- Reduce el tiempo de SLA (Service Level Agreement) del procesamiento. Esto ayuda a las organizaciones a reducir el tiempo del SLA desde días u horas hasta minutos, ya que los datos estarían preparados para ser consumidos por otras aplicaciones en mucho menos tiempo.
- Reduce la carga de trabajo en el mantenimiento del *pipeline*.
- Administra fácilmente la actualización y eliminación de datos. Esto es algo muy beneficioso para casos de uso muy típico en organizaciones como son por ejemplo controlar el GDPR,

gestión de sesiones, gestión de datos duplicados en sus datos, capturar los cambios en los datos, etc.

- Reducción del coste de la infraestructura con un cómputo y almacenamiento elásticos e independientes.

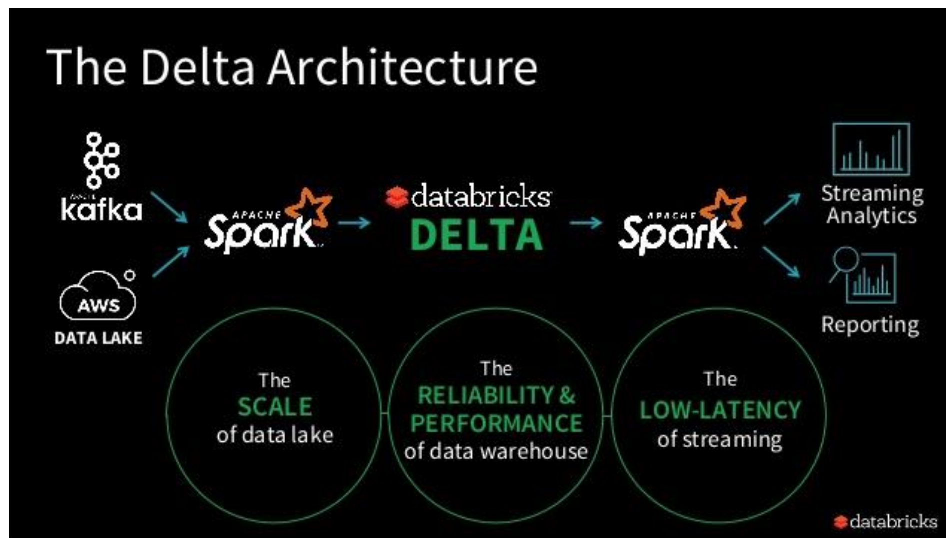


Figura 42. Ventajas de usar la arquitectura Delta. [22]

6. Traffic Monitoring. La plataforma de datos

6.1 Introducción, ¿qué es Traffic Monitoring?

Traffic Monitoring se trata de un sistema de procesamiento en la Nube cuyo objetivo es tener la capacidad de procesar grandes cantidades de datos provenientes de sensores de tráfico instalados en las carreteras belgas manteniendo unos niveles de latencia bajos para poder mantener los informes del estado de las carreteras y de los sensores lo más cercano a tiempo real. En adición a este objetivo el sistema también tendrá el objetivo de realizar mediciones acerca del número de personas que se exponen a impactos publicitarios en estas vías de circulación.

Aunque los objetivos del sistema ***Traffic Monitoring*** sean estos, el objetivo principal del desarrollo de este sistema gira en torno a cómo se desarrollan estos tipos de sistemas y plataformas de procesamiento de datos en la nube, qué tecnologías y arquitecturas se emplean para construirlas y cuáles son sus beneficios, así como demostrar que un sistema de este calibre está más cerca del mundo cotidiano de lo que parece.

De esta forma, con ***Traffic Monitoring*** se ha sido capaz de mostrar un ejemplo de trabajo en la nube desarrollando una plataforma de datos para una empresa o gobierno, simulando un proyecto ficticio, con el que se intenta dar solución también a un problema ficticio, realizando una simulación de caso real. De esta misma forma, se han expuesto en este proyecto tecnologías, herramientas, servicios y arquitecturas que están siendo utilizadas ahora mismo por empresas multinacionales como es el caso de Spark utilizado por empresas como ***Netflix, Amazon, Alibaba, etc***. o Delta Lake utilizado por empresas como ***McAfee, Alibaba, Comcast o Ebay*** por ejemplo entre otras muchas.

La idea de desarrollar este proyecto vino a partir de acordar con el director del TFG, José Joaquín Cañadas, que me gustaría desarrollar un trabajo fin de estudios que estuviera orientado al mundo del Big Data y el procesamiento de datos a gran escala. A partir de ahí, estuvimos buscando ideas prácticas para poder plasmar este objetivo sobre un ejemplo y proyecto práctico, que es donde mejor se ven este tipo de objetivos y tecnologías, cuando entran en acción en un proyecto real.

De esta forma, llegamos a descubrir que existía una fuente de datos en streaming gratuita publicada por el gobierno belga, que exponía los datos de los sensores que tienen instalados en sus carreteras. Estos datos son actualizados minuto a minuto, y cada archivo se trataba de un XML de 180000 líneas y datos de alrededor de 4500. Al ver el volumen de datos, pensamos que sería un buen ejemplo sobre el que desarrollar este proyecto.

6.2 Modelo de datos. Esquema en estrella

6.2.1 ¿Qué es?

El modelado de datos a la hora de diseñar un flujo de datos es algo muy importante. El objetivo de diseñar un buen modelo de datos es organizar los datos que va a contener nuestro sistema de forma conveniente y eficiente. Un buen diseño de datos debe estar centrado en la eficiencia tanto de cómputo como de costes de mantener ese modelo de datos, así como en la utilidad que el modelo de datos ofrece.

El esquema en estrella [38] es una de las soluciones a los modelos de datos utilizados en los flujos de datos ETL. El objetivo principal de este modelo de datos es simplificar la tarea de crear un modelo, siendo capaz de crear un modelo de datos simple y fácil de entender optimizado para hacer peticiones a los datos. El proceso de creación de un modelo de datos en estrella pasa por ver cuáles son las características más esenciales de nuestros datos. Estas características más esenciales o básicas se denominan dimensiones. El conjunto de varias dimensiones en una sola tabla se denomina hecho. Se puede ver la estructura que suele tener un modelo en estrella en la figura 43.

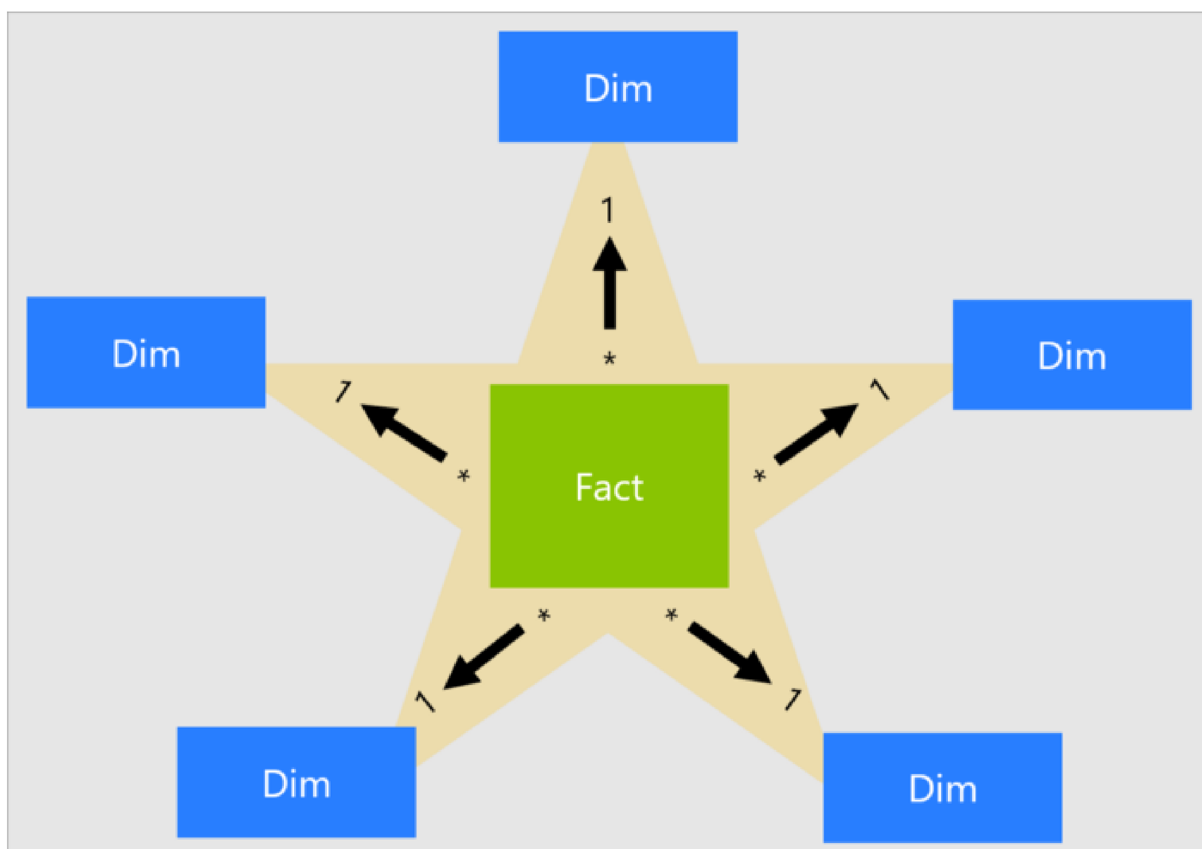


Figura 43. Esquema en estrella.

6.2.2 Tablas de dimensiones

Ahora que ya sabemos que son las tablas de dimensiones, vamos a ver cuáles serían las tablas de dimensiones de nuestro sistema. En nuestro caso, identificar las tablas de dimensiones será muy sencillo.

Por un lado, tenemos una fuente de datos que se encarga de describir información relacionada con los sensores. Entre esta información contamos con la localización, fecha de última actualización, unidad de procesamiento, identificador, etc. Esta fuente de datos es en sí una tabla de dimensión, la cual llamaremos *sensor_description* y que contendrá información sobre nuestro sensor.

Por otro lado, tenemos la fuente de datos que nos informaba sobre mediciones de los sensores. De esta fuente de datos vamos a extraer otra dimensión. La unidad mínima de información relevante de esta fuente de datos es la que nos dice el volumen de vehículos de un tipo y la velocidad media que llevaban en un determinado punto en el tiempo. Esta dimensión se llamará *sensor_data*.

En la parte de procesamiento que vamos a realizar, no necesitaremos más datos que los contenidos en estas dimensiones para conseguir el objetivo que buscamos así que al momento de desarrollar este proyecto contaremos tan solo con estas dimensiones.

6.2.3 Tablas de hechos

Ahora que ya sabemos las dimensiones con las que contamos, vamos a ver cuáles serían las tablas de hechos que necesitamos definir a través de ellas.

En primer lugar, necesitaremos una tabla de hecho con la que se puedan explicar sucesos que hayan tenido lugar. La primera tabla de hechos que vamos a construir consiste en una tabla en la que podamos ver la cantidad de tráfico que ha circulado por un punto en un espacio de tiempo determinado, de forma que podamos saber también la localización en la que ocurrió esa medición. Por tanto, la primera tabla de hechos estará formada por la agrupación en forma horaria y por día del tráfico que haya tenido lugar. Esta tabla de hechos se utilizará en conjunto con el procesamiento en *streaming* para comparar la cantidad de tráfico durante esa hora con la cantidad de tráfico durante esa hora durante la última semana..De esta forma, contamos con las medidas tomadas en la dimensión de *sensor_data* y la localización e información del sensor de la dimensión *sensor_description*.

En segundo lugar, necesitaremos tener una dimensión que nos diga cuántas personas como máximo han circulado por un punto durante un espacio de tiempo determinado, en este caso una hora. Para ello tendremos que contabilizar la cantidad de vehículos que han circulado por cada sensor durante una hora y contar las personas según el número máximo de personas que puede ir en cada tipo de vehículo, extrayendo la localización por la que han pasado, es decir, del sensor, de la dimensión de *sensor_description* y los datos del número de vehículos de la dimensión de *sensor_data*.

Estas serían las dos tablas de hechos que tenemos en nuestro modelo de datos (ver Figura 44).

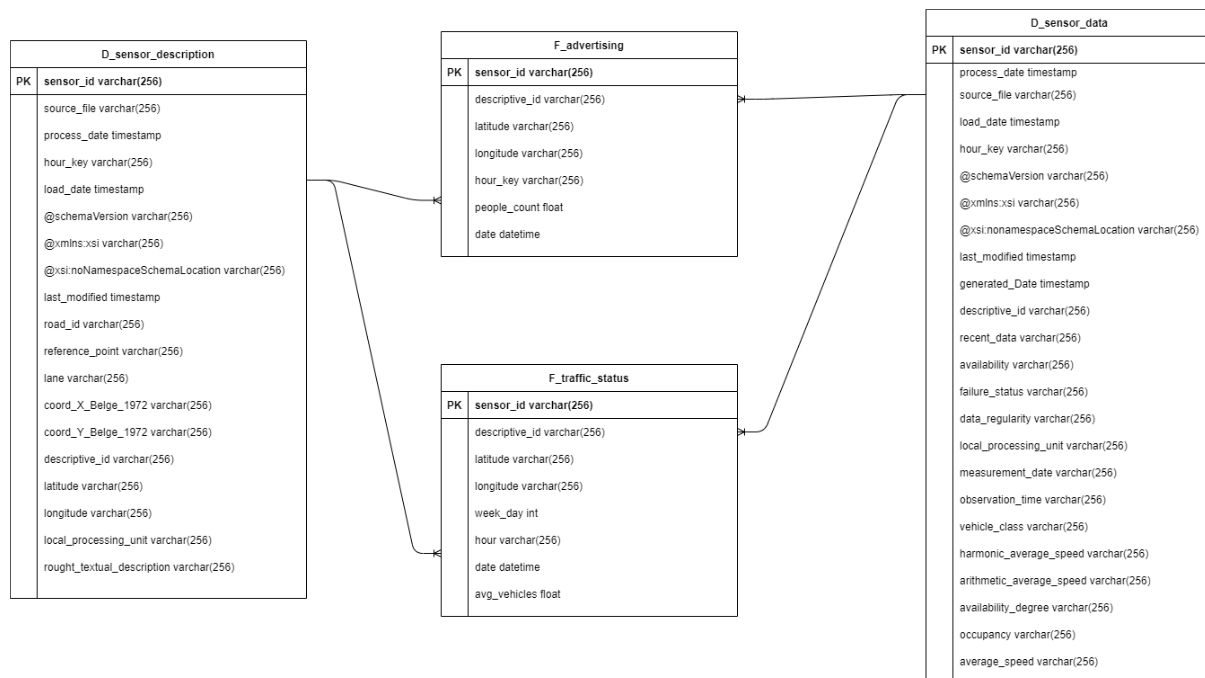


Figura 44. Modelo de datos.

6.3 Ingestión de los datos

Como ya vimos en la solución propuesta, la ingesta de datos la vamos a realizar utilizando el servicio de Azure llamado Azure Functions.

Dentro del servicio de las Azure Functions contamos con una capa gratuita que nos permite hasta un millón de ejecuciones sin coste alguno. Tendremos que realizar una llamada por minuto a cada una de las URLs de la API para obtener los datos. Una llamada será para obtener los datos relacionados con la descripción y características del sensor. La segunda llamada sería para obtener las mediciones tomadas por el sensor.

En total realizaremos una suma de 86400 llamadas a la API y un total de 43200 ejecuciones de nuestra Azure Function para conseguir los datos. Por lo tanto, el proceso de ingestión de datos, de momento será gratuito.

Vamos a ver ahora cada una de las partes en las que se divide nuestra Azure Functions para conseguir los datos de la API.

6.3.1 Entradas de datos

Nuestra Azure Function contará con una única entrada de datos. Como ya vimos, una Azure Function tiene que contar como mínimo con un parámetro de entrada que será el punto de entrada para ejecutar nuestra función.

En este caso contaremos con un *trigger* por tiempo, para que nuestra función se ejecute una vez por minuto. Esto se lo especificamos a través de una expresión de *cron.* Cron se trata de un administrador de procesos en segundo plano que ejecuta procesos en intervalos de tiempo regulares.

En el archivo *function.json* de nuestra función, configuraremos un *binding* de entrada. Los *binding* pueden ser tanto de entrada como de salida. Para nuestra función tendremos que definir más *bindings* pero de momento vamos a ver cómo sería nuestro *binding* de entrada.

```
{
  "name": "mytimer",
  "type": "timerTrigger",
  "direction": "in",
  "schedule": "0 * * * * *"
},
```

Figura 45. Azure Function. Input Binding.

En la Figura 45 podemos ver como se especificaría el *binding* de tiempo de entrada. Un *binding* consta de un mínimo de 3 atributos:

- Nombre: Es el nombre con el que identificamos a nuestro parámetro dentro de la función.
- Tipo: Se trata del tipo de Binding
- Dirección: Identifica que tipo de conexión se trata. Puede ser de entrada o salida.

Podemos ver entonces en la Figura 45 como nuestro *binding* de tiempo se llama “myTimer”, es de tipo “timerTrigger” y que se trata de un parámetro de entrada. Al tratarse de un *trigger* por tiempo, podemos ver como hemos especificado la expresión de cron en el atributo *schedule*. Con esa expresión le estamos diciendo a nuestra función que deberá ejecutarse una vez por minuto.

6.3.2 Obteniendo los datos

Una vez que tenemos definido cómo y cuándo se va a lanzar la función, vamos a especificar la lógica de nuestra función para obtener los datos de la API.

En primer lugar, podremos fijarnos que en ningún lugar de nuestro código aparece ninguna URL a ninguna API. Esto es porque Las Azure Functions cuentan con unas variables llamadas variables de entorno, a través de las cuales podemos definir variables que podrían comprometer nuestro sistema, como podrían ser contraseñas o cadenas de conexión a otros servicios.

En nuestro caso, hemos definido tanto las URLs de la API como la cadena de conexión para luego guardar los datos descargados de la API en estas variables de entorno. (ver Figura 46).

```
# variables
connect_str = os.environ['AzureWebJobsStorage']
sensor_data_url = os.environ['sensor_data_url']
sensor_description_url = os.environ['sensor_description_url']
```

Figura 46. Obteniendo las variables de entorno.

Estas variables de entorno las tendremos que definir también en la configuración de nuestra Azure Function, Para ello tenemos que ir a la parte de Configuración de en Ajustes, en el panel izquierdo de nuestra Azure Function. En la Figura 47 podemos ver las variables configuradas de nuestra Azure Function y encontrar las 3 variables que hemos necesitado usar para la lógica de nuestra función.

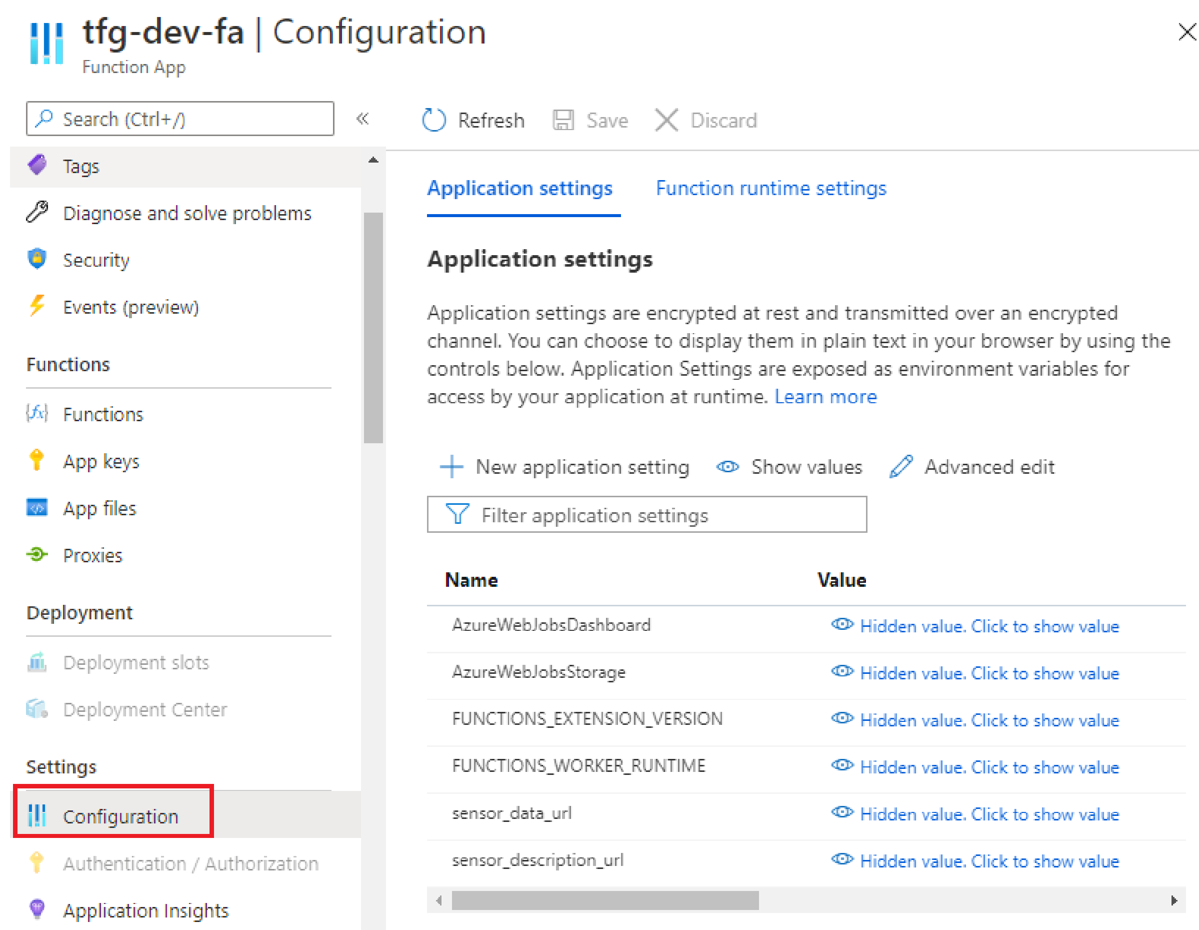


Figura 47. Configuración de la Azure Function.

A la hora de conseguir los datos de la API, lo haremos con una simple llamada a la API. Como estoy usando Python, yo he hecho uso de la librería requests para ello. Requests no viene instalada por defecto en los entornos de Python, así que tendremos que añadir esta librería al archivo de requisitos de nuestra Azure Functions. Este archivo lo encontraremos en la carpeta raíz de la Azure Function y se llamará "*requirements.txt*". Aquí tendremos que añadir todas aquellas librerías que no vienen por defecto instaladas como parte de Python.

Uno de los problemas con los que nos encontramos al bajarnos los datos, es el propio formato de los datos. Los datos se encuentran en formato XML (*Extensible Markup Language*). El problema que tenemos con este formato es que cada archivo que guardemos tendrá un tamaño aproximado de 8.4 megaBytes. Si hacemos cuentas, tenemos que guardar 2 archivos por minuto de este tamaño, teniendo al cabo de una hora un tamaño de datos ingerido de entorno a algo menos de 1 GigaByte. Por lo que al día estaríamos ingiriendo algo menos de 24 GigaBytes.

Entonces tocaba pensar y poner una solución a este problema. La primera solución pasó por convertir los archivos XML a archivos JSON. Con esta primera solución se consiguió reducir el tamaño del archivo de 8.4 MegaBytes a un tamaño de unos 4 MegaBytes. Esta primera solución, combinándola con una compresión de tipo gzip. Gzip se trata de una abreviatura de GNU Zip y se trata de un tipo de compresión más potente que el típico Zip. De esta forma se consiguió reducir el archivo de un tamaño inicial de 8.4 MegaBytes a un tamaño de 119 KiloBytes. Todo esto sin perder ni un solo dato de los recibidos.

Por lo tanto, utilizando esta solución se consiguió un ahorro de espacio y transferencia de archivos, pasando de descargar y almacenar casi 1 GigaByte por hora a tan solo 14 MegaBytes por hora.

Hay que decir también que la solución de transformar los archivos a JSON, no es la mejor técnica, debido a que estos archivos deberían mantenerse en el formato tal cual están siendo generados. Pero esto era necesario para poder realizar próximos procesamientos, ya que Stream Analytics no es capaz de procesar archivos con formato XML. De esta forma, se tuvo que cambiar el formato de los archivos a JSON.

De esta forma, el proceso para obtener los datos pasará por realizar la llamada a la API, transformar el archivo de XML a JSON y comprimirlo.

Todo el proceso estará dentro de un *Try/Catch* de forma que fallase algo durante la obtención y preparación de los datos para ser guardados, podremos saberlo. No porque la función falle, sino porque todo el proceso tendrá un log (ver Figura 48) donde podremos ver en todo momento qué datos nos estamos bajando y que datos no han podido ser obtenido por el motivo que sea. La tabla de log identificará de forma independiente cada una de las llamadas a la API, por lo que puede fallar una de ellas, y no afectar en nada a la otra llamada.

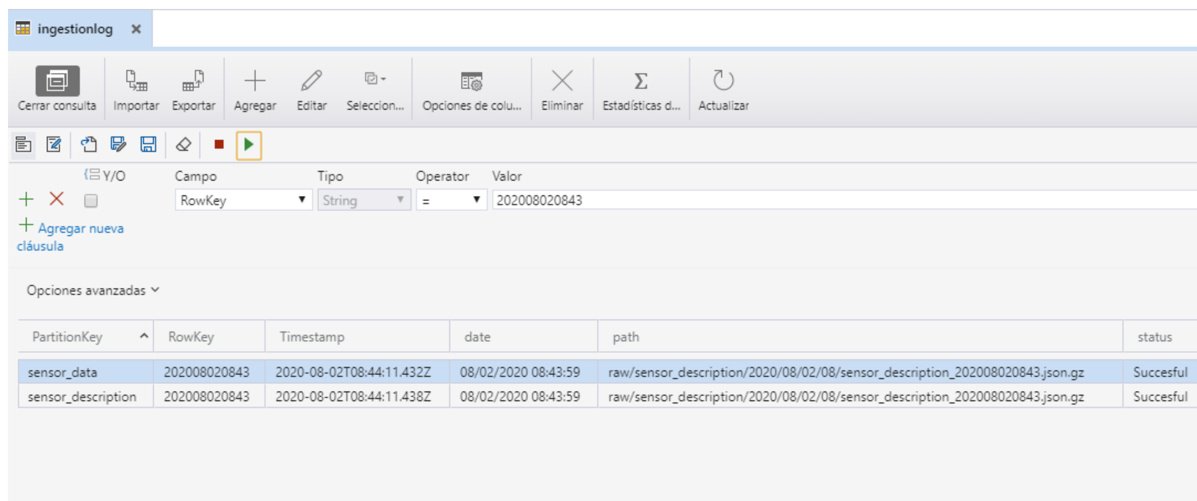


Figura 48. Log de la ingestión de datos.

6.3.3 Almacenando los datos obtenidos

Una vez procesados los datos y listos para ser escritos utilizaremos una conexión hacia nuestro *Storage account* para subir nuestros datos. Subir los datos será muy sencillo. Haremos uso de la API de Storage Account y de la librería de python para subir los datos. La librería es ***azure.storage.blob***.

Haremos uso también de la librería StringIO para simular un archivo en nuestra función, el cual una vez comprimido lo subiremos a nuestro Data Lake, dentro de una estructura de directorio tal que identificamos la fuente de datos, el año, mes, día y hora en la que el archivo fue cargado, y todo esto irá dentro de una carpeta llamada *RAW*. Dentro de esta carpeta será donde incluiremos todos nuestros archivos descargados directamente de API.

6.4 Procesamiento de los datos en *Databricks*

Vamos a ver ahora como se ha implementado el procesamiento de datos en Databricks. En este servicio es donde hemos implementado parte de nuestra arquitectura Delta. Vamos a ver ahora el procesamiento realizado en las diferentes capas, así como detalles curiosos de la implementación.

6.4.1 Procesando la capa de Landing

El objetivo de esta capa es construir nuestra primera tabla de Delta a partir de los datos que tenemos guardados en nuestra capa de Raw. En este primer procesamiento haremos un *streaming* de archivos a delta, por lo tanto tendremos que manejar todos y cada uno de los ficheros que tenemos que leer.

Para tener que evitar tener que pasarles nosotros manualmente los ficheros, o tener que manejar nosotros los archivos que se van a leer, vamos a hacer uno de un conector que implementa Databricks llamado *abs-aqs* (Azure Blob Storage - Azure Queue Storage). De esta forma, lo que tenemos es un Storage Queue que gracias a un Azure Event Grid, guardaremos en la cola el *path* a cada uno de los archivos que llegan a nuestra capa de raw en el Data Lake (ver Figura 49). A la hora de leer con *abs-aqs* tendremos que especificar la cola de la que queremos consumir los mensajes (Cada mensaje contendrá

el path a un archivo) así como la cadena de conexión para conectarnos al Storage Account donde se encuentra nuestra cola. Además como aspecto importante, hay que destacar también que tendremos que especificar el esquema de nuestro archivo de lectura. En este caso, vamos a leer los archivos en modo texto. Leer en modo texto nos permitirá coger todo el contenido del archivo, transformarlo en una cadena de texto enorme con todo el contenido y lo añadirá a una columna llamada value.

```
spark.readStream
.format("abs-aqs")
.option("fileFormat", "text")
.option("queueName", "sensor-description-queue")
.option("connectionString", conn_datalake)
.schema(StructType([StructField("value", StringType())]))
.load()
```

Figura 49. Leer Stream con Structured Streaming utilizando abs-aqs.

Una buena práctica, que utilizaremos nosotros también es añadir a nuestro *dataframe* de entrada una columna, la cual nos diga de cuál archivo proviene ese dato. Esto es muy común de hacer y recomendable para que en caso de que alguna fila venga con datos corruptos, podamos echar un vistazo a los archivos de entrada para comprobar que los datos que vienen en ese archivo vengan bien.

Otro aspecto a destacar, será el hecho de agregar una columna que nos diga la hora y un día en concreto del año. Esto es que tendremos una columna que nos dirá que un registro pertenece por ejemplo a la hora 14 del día 15 de julio de 2020. En este caso, el resultado de esta columna sería 2020071514. Este número será usado como clave para poder particionar nuestra tabla de delta de forma horaria con el objetivo de mejorar el rendimiento y latencia de nuestra tabla a la hora de realizar operaciones sobre ella.

Una vez leídos nuestros datos y añadidas las columnas necesarias para poder particionar y poder tener una columna para rastrear posibles fallos en los archivos, vamos a escribirlos a nuestra primera tabla de Delta. (ver Figura 50).

```
query = (
  reader
  .writeStream
  .outputMode("append")
  .foreachBatch(
    lambda batch_df, batch_id:
    merge(
      batch_df,
      batch_id,
      sink_path = landing_path + "delta_table/",
      merge_columns = ["load_date"],
      partition_column = "hour_key"))
  .option("checkpointLocation", landing_path + "checkpoint/")
  .trigger(once = True)
  .start()
).awaitTermination()
```

Figura 50. Escribiendo stream con Structured Streaming.

Al escribir nuestro stream, tenemos varios aspectos a destacar y bastante curiosos. En primer lugar, si vamos leyendo de arriba hacia abajo en la Figura 50, podemos ver cómo estamos utilizando una función merge que hemos definido nosotros. A la función foreachBatch le tenemos que pasar como parámetro una función donde haremos distintas operaciones con los datos que hemos leído. En nuestro caso, utilizaremos la función merge del foreachBatch para realizar las operaciones *upsert* sobre nuestro delta. A nuestra función le pasamos diferentes elementos. Entre ellos están el *dataframe* que queremos introducir a nuestro delta, el identificador del *dataframe*, el directorio donde se encuentra nuestro delta de destino, las columnas por las que realizaremos nuestro *upsert* y por último las columnas por las que vamos a particionar. En la figura 51 podemos ver cómo está implementada nuestra función merge.

Tras nuestra función merge, podemos ver cómo están definidos los *checkpoints*. Los *checkpoints*, nos permiten evitar que al tener que volver a leer todo si se ejecuta de nuevo el pipeline, leamos archivos que ya han sido leídos anteriormente.

En este caso, a priori nos daría igual porque la cola se queda limpia de mensajes cada vez que se lee y se escriben los mensajes. Pero nos sirve para poder ejecutar de nuevo el pipeline leyendo de la carpeta directamente en lugar de leer de la cola, de forma que podríamos volver a leer todo de nuevo, o intentarlo y gracias a los *checkpoints* solo leeremos aquello que no se haya leído antes.

La siguiente función que encontramos tras definir los *checkpoints*, se trata de una función *trigger* la cual le decimos que su parámetro *once* sea igual a *True*. Esto se lo especificamos para decirle a nuestro streaming que una vez termine de procesar los datos que ha leído se detenga y no se quede esperando a que lleguen nuevos datos que escribir.
Como último aspecto a destacar, tenemos al final de nuestro proceso de escritura un *awaitTermination().* Esta función se utiliza para asegurarnos de que el *notebook* o *cluster* no dejará de ejecutarse hasta que el streaming acabe de escribirlo todo.

```
def merge(batch_df, batch_id, sink_path, merge_columns, partition_column):
  """
  Merge a batch from the streaming with the sink data source based on columns parameters.
  The sink dataset can be partitioned too.
  """
  # Check if delta table exists
  try:
    sink = DeltaTable.forPath(spark, sink_path)
  except:
    if partition_column is None:
      batch_df.write.format("delta").save(sink_path)
      return
    else:
      batch_df.write.format("delta").partitionBy(partition_column).save(sink_path)
      return

  # Define conds
  conds = " and ".join([f"source.{col} = sink.{col}" for col in merge_columns])

  # Execute merge
  (
    sink.alias("source")
      .merge(
        batch_df.alias("sink"),
        conds
      )
      .whenNotMatched().insertAll()
  ).execute()
```

Figura 51. Función merge para escribir un stream.

Un aspecto destacable de nuestra función de merge es que es capaz de lidiar con la escritura cuando el delta no existe. Esta característica la añadí para cuando se trate de la primera ejecución, en lugar de intentar ejecutar el *upsert* (el cual fallará porque aún no existe el delta) lo que hará será escribir los datos directamente crean la tabla de delta.

Tanto la capa de Landing para las mediciones del sensor como para los datos que describen el sensor serán prácticamente iguales. Las únicas diferencias estarán en el origen y destino de los datos.

6.4.2 Preparando la capa de Staging

Ahora que ya hemos dado forma a la capa de landing, vamos a proceder al procesamiento de los datos. Durante esta capa daremos a cada campo el tipo de dato que se trata. Para una cadena de texto le daremos el formato de *String*, para una fecha le daremos formato de *timestamp* o para un número le daremos formato de *float* o *long* en función de cómo sea el dato.

A parte de dar formato a nuestros datos, también tendremos que darle forma, ya que en la capa de landing se encuentran totalmente sin estructura. El esquema que presentan los datos ahora mismo en la capa de Landing se puede ver en la figura 52.

```
reader: pyspark.sql.dataframe.DataFrame
    value: string
    source_file: string
    process_date: timestamp
    load_date: timestamp
    hour_key: string
```

Figura 52. Esquema tabla de Landing.

Dentro del campo *value* es donde podremos encontrar todos los datos leídos. Estos datos ahora mismo son una simple cadena de texto enorme de la cual no podemos procesar nada aún.

Por lo tanto, tanto en la capa de staging de los datos que describen el sensor como la de las mediciones del sensor, tendremos que darles forma a estos datos. Para ello, vamos a necesitar hacer uso de 2 herramientas

Por un lado, vamos a necesitar extraer o construir el esquema de los datos que tenemos en nuestro campo *value.* Para hacer esto, podemos hacerlo de dos formas diferentes. La primera consiste en construirlo nosotros manualmente, asignando el nombre de cada campo y el tipo de dato que es. La segunda forma es más rápida aunque precisa de una revisión de los campos posteriormente para ver si algún dato no tiene el tipo de dato esperado.

El esquema es una parte importante de nuestros datos, porque aquí es donde describimos como es nuestro *dataframe*, es donde decimos los campos que contienen nuestros datos así como su naturaleza. El esquema lo podemos construir manualmente haciendo uso de la librería de tipos que contiene *Pyspark.* Aunque en nuestro caso vamos a inferir el esquema automáticamente. Para poder inferir el esquema automáticamente vamos a necesitar un pequeño ejemplo de nuestros datos, donde se pueda ver perfectamente cuales son los campos contenidos en nuestros datos. En nuestro caso, se tratan de archivos JSON por lo que el ejemplo de los datos será un pequeño JSON del cual extraemos el esquema. Para extraer el esquema haremos uso del *SparkContext*, para construir una estructura de datos a través de nuestro JSON de ejemplo y posteriormente leeremos esta estructura de datos como si de un conjunto de datos normal se tratase en *Spark*, y con el atributo *schema* extraemos el esquema inferido automáticamente por Spark. (ver Figura 53).

```
1 schema = """
2 {"mivconfig": {"@xmlns:xsi": "http://www.w3.org/2001/XMLSchema-instance",
  "@xsi:noNamespaceSchemaLocation": "http://miv.opendata.belfla.be/miv-config.xsd",
  "@schemaVersion": "1.0.0", "tijd_laatste_config_wijziging": "2019-12-05T10:26:49+01:00",
  "meetpunt": [{"@unieke_id": "3640", "beschrijvende_id": "H291L10", "volledige_naam":
  "Parking Kruibeke", "Ident_8": "A0140002", "lve_nr": "437", "Kmp_Rsys": "94,695",
  "Rijstrook": "R10", "X_coord_EPSG_31370": "144474,5297", "Y_coord_EPSG_31370":
  "208293,5324", "lengtegraad_EPSG_4326": "4,289731136", "breedtegraad_EPSG_4326":
  "51,18460764"}]}}
3 """
4 df = sc.parallelize([schema])
5 schema = spark.read.json(df).schema
```

Figura 53. Inferir esquema de datos automáticamente.

En caso de que la estructura de nuestros datos cambie en algún momento, solo tendremos que cambiar la estructura de nuestro JSON de ejemplo aquí, y fusionar el esquema de nuestro datos históricos junto con los datos nuevos, y de esta forma podremos seguir conservando todos los datos sin necesidad de volver a reescribir la tabla o cambiar de directorio para escribir los nuevos datos.

Ahora que ya tenemos nuestro esquema, podremos dar formato de forma rápida a nuestros datos contenidos dentro de la columna *value*. Para hacer esto vamos a hacer uso de una función de Spark llamada *from_json()*. Esta función recibe dos parámetros de entrada. El primer parámetro se trata del nombre de la columna que contiene nuestros datos en formato JSON. El segundo parámetro se trata del esquema que estos datos en formato JSON tienen. Lo que esta función se encarga de hacer es dar formato a los datos contenidos en la columna que nosotros le pasemos de nuestro *dataframe* para aplicarle el esquema que le hemos pasado. En caso de que haya algún campo extra, este será ignorado. En el caso de que uno de los campos de nuestro esquema no se encuentre en el archivo JSON, su valor será igual a nulo.

Una vez llegados a este punto, ya tenemos hecho uno de los primeros pasos de nuestro procesamiento en Staging. Nuestros datos ya cuentan con formato y podemos pasar a deserializar los datos contenidos dentro de la columna *value*.

A la hora de limpiar los datos, va a ser necesario limpiar los datos, renombrar columnas (ya que los nombres de las columnas están en belga y más difícil trabajar con estos nombres, así que les daremos nombres que nos sean más amigables para trabajar con ellos), y cambiar el tipo a algunas columnas que no hayan sido formateadas de la forma correcta.

Para cada una de las fuentes de datos tendremos que definir una serie de transformaciones para limpiar los datos. Entre las funciones más destacable y útiles a la hora de realizar este proceso tenemos:

- Explode: Esta función es muy común usarla cuando tenemos un elemento que contiene un campo que se trata de un array y no queremos mantener un array en nuestro *dataframe*. En este caso lo que queremos es que se genere una nueva fila por cada elemento dentro del array manteniendo el resto de valores igual. Con la función *explode* somos capaces de conseguir esto fácilmente.
- withColumn: Esta función es usada para añadir o transformar valores dentro de un *dataframe*. Es muy común utilizar esta función junto con la función *explode*, con el objetivo de que los valores extraídos por la función *explode* se mantengan dentro del campo con el que queramos nombrar estos valores. A parte de esto, podemos añadir nuevas columnas a nuestro *dataframe* que sean calculadas a través de otras columnas o para rellenar una columna con valores predeterminados para ser usados más adelante.
- withColumnRenamed: Esta función se utiliza para renombrar columnas y sustituir su nombre actual por un nuevo nombre.
- cast: Esta función se utiliza sobre una columna para cambiar el tipo de dato contenido dentro de ella. Nosotros lo utilizaremos principalmente para cambiar el tipo de dato de algunas fechas de *string* a *timestamp*.

Para realizar las transformaciones en nuestro *dataframe* vamos a hacer uso de la función *transform.* Esta función se utiliza para dividir la lógica de transformaciones sobre nuestro *dataframe* y poder así

mantener la legibilidad dentro de nuestros notebooks. Un *dataframe* puede sufrir tantas transformaciones como sean necesarias.

Una vez tengamos nuestros datos transformados, los escribiremos a su correspondiente tabla de delta según la fuente de datos de la que se trate. Cada una de las tablas salientes de nuestro procesamiento en *staging* dará lugar a una tabla de dimensión. Contaremos con dos tablas de dimensiones: *sensor_description* y *sensor_data.*

6.4.3 Capa de Agregación: Estado del tráfico

En esta capa vamos a procesar y agrupar los datos que tenemos limpios y preparados para ser utilizados y conseguir información relevante con el objetivo de poder extraer información relevante de ellos y poder actuar en consecuencia. Esta es la denominada capa de oro, o *curated.*

La tabla del estado del tráfico será una de nuestras tablas de hechos. En esta tabla encontraremos información relevante sobre el estado del tráfico en cada punto durante la última semana a nivel horario y por día. Contaremos con una media del tráfico por cada punto durante la última semana.

La lógica relacionada con este cálculo se hará dentro de la parte de *batch* de nuestro stream durante la fusión de los datos de entrada con los datos que podamos hallar dentro de nuestra tabla. A priori no deberían de introducirse actualizaciones dentro de los datos que ya existan, ya que cada día se incorporarán nuevos datos que contendrán la información desde el día de ejecución hasta 7 días antes.

Aun así, hay que tenerlo en cuenta, ya que podría darse el caso de que algún día no se procese bien y haga falta reprocesar datos o cambie algo en las tablas de dimensiones de staging, e hiciese falta actualizar valores.

El motivo por el cual la lógica se encuentra en ese lugar y no fuera de él es que las operaciones que se pueden realizar entre *streams* están limitadas. De esta forma, no se pueden realizar operaciones entre agrupaciones de datos entre dos stream al igual que no se pueden llevar a cabo todos los tipos de uniones entre streams.

Es por eso que, para obtener el resultado de la tabla de hechos del estado del tráfico, se implementa un nuevo merge específico para esta tarea. Este nuevo merge contiene lo mismo que el merge normal, con el cambio de que implementa un paso donde se realizan las transformaciones.

Para obtener el resultado se utilizan las tablas de dimensiones de *sensor_description* y *sensor_data*. Podemos ver la lógica en la figura 54. Lo que se hace en esta lógica en primer lugar es obtener para cada sensor, los datos sobre su localización más actualizados. La última versión de su descripción. En segundo lugar, filtramos los datos contenidos en la tabla de *sensor_data* para quedarnos solo con la última semana de datos y así poder trabajar solo con ellos. Añadiremos a estos datos dos columnas para identificar el día de la semana en el que se cargaron los datos y a los que pertenece la medida y otra columna para saber a qué hora hace referencia. Estas dos columnas nos ayudarán a poder agrupar los datos y así poder realizar la media de vehículos que suelen pasar por un punto en la carretera durante un espacio de tiempo determinado. Este sería el siguiente paso, en el cual agrupamos los datos que hemos filtrado de la tabla de *sensor_data* y calculamos la media de vehículos. Una vez calculados los

datos que queremos, se realiza un *join* entre el primer *dataframe* del cual obtuvimos los datos más actualizados con las configuraciones y descripciones de cada sensor y el segundo *dataframe* el cual contiene los datos que necesitamos saber para luego nuestros informes. Una vez tenemos este resultado se escribe a la capa de curated y está listo para ser escrito a la base de datos.

```
1  def traffic_status_logic(df):
2    D_sensor_description = (
3      spark
4      .read
5      .format("delta")
6      .load(source_sensor_description_path)
7    )
8
9    updated_sde = (
10     D_sensor_description
11     .groupBy("sensor_id", "descriptive_id")
12     .agg(
13       F.max(F.col("load_date")).alias("date"),
14       F.first(F.col("latitude")).alias("latitude"),
15       F.first(F.col("longitude")).alias("longitude")
16     )
17   )
18
19   filtered_traffic_df = (
20     df
21       .filter(F.col("load_date") > F.date_sub(F.col("load_date"), 7))
22       .withColumn("week_day", F.dayofweek(F.col("generated_date")))
23       .withColumn("hour", F.date_format(F.col("generated_date"), 'H'))
24   )
25
26   traffic_mean = filtered_traffic_df.groupBy("sensor_id", "descriptive_id", "week_day",
   "hour").agg(F.avg(F.col("vehicles_number")).alias("avg_vehicles"))
27
28   traffic_status = updated_sde.join(traffic_mean, ["sensor_id", "descriptive_id"])
29
30   return traffic_status
```

Figura 54. Lógica para calcular el estado del tráfico.

6.4.4 Capa de Agregación: Estadísticas de afluencia de gente

Al igual que en el anterior punto, vamos a ver también cómo hemos llegado a construir esta tabla de hechos para poder extraer conclusiones de nuestros datos acerca de la afluencia y movimiento de la gente por las diferentes vías.

Para calcular esta tabla de hechos lo que haremos será calcular a nivel horario la cantidad de personas que circulan por un punto determinado de una carretera. Cada punto a medir será la localización de un sensor.

Al igual que ocurre con la lógica hecha para calcular el estado del tráfico, esta lógica también se ha hecho dentro de la función del merge debido a los tipos de operaciones que hay que hacer para poder calcular, por lo que también cuenta con una función merge particular.

Uno de los aspectos que se ha de tener en cuenta de la implementación de esta parte es el mapeo que se realiza para poder establecer el número de personas por típico de vehículo. Esto quiere decir que se realiza una sumatoria total de personas en función de identificador de sensor y hora, y se suman el producto del total de vehículos de cada tipo por el número máximo de personas que puede llevar ese vehículo.

Tenemos 5 tipos de vehículos:

- Tipo 1: Motos o bicicletas. Vehículos de 1 sola plaza.
- Tipo 2: Coches. Vehículos de 5 plazas.
- Tipo 3: Coches grandes familiares o furgonetas. Vehículos de hasta 7 plazas.
- Tipo 4: Camiones o tráileres. Vehículos de mercancías con posibilidad de llevar hasta 4 personas.
- Tipo 5: Autobuses. Vehículos que pueden llevar 50 personas aproximadamente.

Teniendo en cuenta esto, vamos a realizar el cálculo de nuestra lógica. (ver Figura 55). Hay parte de la lógica que será muy similar al cálculo de la tabla de hechos del estado del tráfico en las carreteras como es la parte en la que obtenemos la información más actualizada referente a la descripción y localización de cada sensor.

En los pasos siguientes es donde variará un poco. En primer lugar tendremos que definir una nueva columna. Esta nueva columna será la que nos diga cuántas personas han circulado durante el espacio de tiempo de 1 minuto por un tramo de vía. Esta columna necesitará saber del número total de vehículos que han circulado por un tramo así como del tipo de vehículo que se trataba para hacer uso de un mapa de valores para interpolar los valores y saber el total de personas que han circulado por la vía aproximadamente.

Finalmente, una vez que tengamos esto calculado, uniremos estos datos con los de nuestro primer *dataframe,* que contiene la localización de los sensores para poder tener las medidas y las localizaciones juntas. Con esto ya tendríamos nuestra segunda tabla de hechos.

```
1  def advertising_logic(df):
2    D_sensor_description = (
3      spark
4      .read
5      .format("delta")
6      .load(source_sensor_description_path)
7    )
8
9    updated_sde = (
10     D_sensor_description
11     .groupBy("sensor_id", "descriptive_id")
12     .agg(
13       F.max(F.col("load_date")).alias("date"),
14       F.first(F.col("latitude")).alias("latitude"),
15       F.first(F.col("longitude")).alias("longitude")
16     )
17   ).drop("date")
18
19   mapping = {
20     "1": 1,
21     "2": 5,
22     "3": 7,
23     "4": 4,
24     "5": 50
25   }
26
27   mapping_expr = F.create_map([F.lit(x) for x in chain(*mapping.items())])
28
29   people_df = (
30     df
31       .withColumn("people_count", F.col("vehicles_number") * mapping_expr[F.col("vehicle_class")])
32       .groupBy("sensor_id", "descriptive_id", "hour_key")
33       .agg(
34         F.sum(F.col("people_count")).alias("people_count")
35       )
36     .withColumn("date", F.to_timestamp(F.col("hour_key"), "yyyyMMddHH"))
37   )
38
39   advertising = updated_sde.join(people_df, ["sensor_id", "descriptive_id"])
40
41   return advertising
```

Figura 55. Lógica para calcular el número de personas por punto y hora.

6.5 Procesamiento de los datos en *Streaming*

Vamos a ver ahora cómo se realiza el procesamiento de los datos en streaming para poder tener una presentación rápida de los datos en los diferentes informes.

Para ello vamos a hacer uso del servicio Azure Stream Analytics. Para trabajar con Stream Analytics hemos definido 2 conjuntos de entrada, 1 referencia a la base de datos SQL, 2 conjuntos de datos de salida directamente a Power BI y por último una Query para seleccionar qué datos queremos enviar a Power Bi.

Vamos a ver que se ha hecho un poco más en profundidad.

6.5.1 Conjuntos de datos de entrada

Para poder trabajar con Stream Analytics y empezar a procesar datos lo primero que necesitamos es obviamente los datos. Los datos los vamos a conseguir de nuestro Data Lake. Más concretamente de la capa de Raw, que es donde almacenamos los datos "tal cual" nos los descargamos de la API.

Cuando comenzamos a bajarnos los datos, había un motivo por el cual los datos tenían que ser formateados de XML a JSON. El motivo es que Stream Analytics no acepta como formato de archivo XML y por tanto se optó por transformarlos a JSON.

La forma en que obtendremos los datos desde nuestro Data Lake hasta Stream Analytics es a través de un *trigger* de evento. El evento que se tiene que dar para obtener y procesar un archivo es que un nuevo archivo llegue a un directorio específico que nosotros podemos especificar. En nuestro caso especificamos 2 directorios diferentes. Uno de ellos es para obtener los datos que describen el sensor, y el otro es para obtener los archivos que contienen las mediciones. En la figura 56 podemos ver cómo se filtra el directorio del cual queremos obtener los archivos.

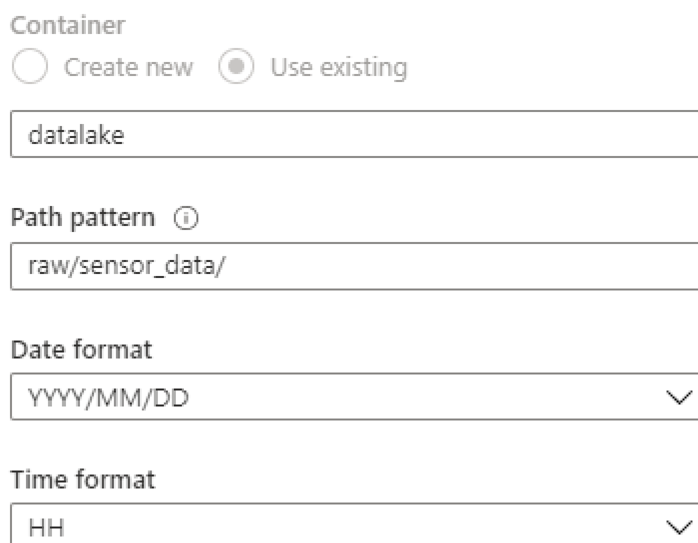


Figura 56. Filtrando directorios para obtener archivos en Stream Analytics.

Esto mismo lo haremos para cada uno de los dos conjuntos de datos que hay que procesar en Stream Analytics.

Por otro lado, se ha establecido una referencia a la tabla de la base de datos que guarda los datos acerca del estado del tráfico, ya que esta tabla será necesaria para comparar el estado del tráfico actual con el estado del tráfico en el pasado haciendo uso de datos históricos para saber si el tráfico es mayor, menor o si está en el intervalo esperado.

6.5.2 Conjuntos de datos de salida

Obviamente los datos que no procesamos no tendrían ningún sentido si estos no se utilizaran para nada. Es por esto que los datos que hemos procesado serán enviados a lo que se conocen como conjuntos de datos en *streaming* de Power BI.

Stream Analytics y Power BI cuentan con una integración interna que nos facilita el poder usar estos datos en Power BI para construir nuestros informes. No tenemos que preocuparnos por vincular Stream Analytics con Power BI ni configurar ningún tipo de clave. Lo único que hará falta será dar un nombre al conjunto de datos, especificar el ambiente de trabajo donde queremos que se encuentren nuestros datos y por último loguearnos con nuestra cuenta de Power BI para conceder permisos de escritura para que Stream Analytics pueda escribir los datos a Power BI.

De esta forma configuraremos dos salidas de datos diferentes, cada una para un conjunto de datos diferente. Tenemos dos conjuntos de datos, uno de ellos es para conocer el estado de los sensores y saber si están funcionando correctamente. El otro conjunto de datos es para conocer el estado del tráfico.

Estos conjuntos de datos se llenarán de datos a través de una query con la que enviaremos los resultados de las consultas sobre los datos de entrada con los resultados necesarios.

Cabe tener en cuenta que Stream Analytics sólo retiene datos de la hora anterior. Es decir, si son las 9 de la mañana Stream Analytics sólo mantiene en memoria datos que sean desde las 9 de la mañana hasta una hora antes. Los datos anteriores se pierden en Stream Analytics.

A nosotros esto no nos afecta ya que para procesar datos históricos ya tenemos el pipeline con databricks donde tendremos igualmente toda esta información almacenada en nuestra Delta Lake.

6.5.4 Query

La query es la parte más importante a la hora de procesar datos en Stream Analytics. Con ella podremos guiar nuestros datos de un conjunto de datos de entrada a un conjunto de datos de salida, realizando por medio las transformaciones necesarias a nuestros datos.

La query que tenemos en nuestro Stream Analytics cuenta con dos sub queries ya que tenemos 2 datasets de salida. En la figura 57 podemos ver los recursos con los que contamos en nuestra query para trabajar con ellos.

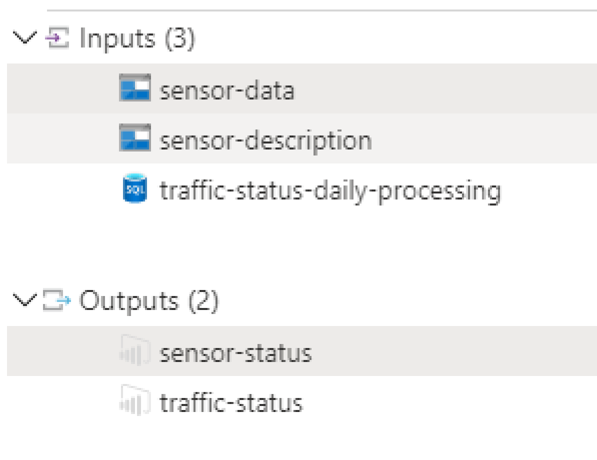


Figura 57. Conjuntos de datos de entrada y salida en Stream Analytics.

Como bien sabemos, tenemos dos conjuntos de datos de salida. El primer conjunto de salida hace referencia al estado de funcionamiento de los sensores. Este conjunto de datos nos permitirá ver si un sensor está o no enviando información y en caso de que lo esté haciendo si esta información que está enviando puede o no presentar fallos, retrasos en el envío de los datos o alguna otra irregularidad.

Para formar este conjunto de datos, en nuestra primera query tendremos que deserializar los datos contenidos tanto en el conjunto de datos de entrada de *sensor-data* como en el de *sensor-description*. Para ello haremos uso de funciones presentes en T-SQL el lenguaje extendido de SQL que utiliza Stream Analytics. La función clave para hacer esto es *GetArrayElements*. Haciendo uso de esta función más un C*ross Apply* de nuestro conjunto de datos junto con los elementos extraídos con la función *GetArrayElements*. (ver Figura 58).

```
SELECT
    [sde].*,
    [meetpuntAlias_de].[ArrayValue] as [MeetPunt_de]
FROM
    [sensor-description] as sde
CROSS APPLY GetArrayElements(mivconfig.meetpunt) AS [meetpuntAlias_de]
```

Figura 58. Deserializando elementos en Stream Analytics.

Una vez hecho esto con ambos conjuntos de datos, uniremos ambos conjuntos por los campos identificados de los sensores, el identificador del sensor y el identificador descriptivo del sensor y nos quedaremos con los campos que más nos interesen de cada conjunto de datos. En este caso nos quedaremos con los campos que nos indican la frecuencia de actualización y el estado de las notificaciones de los sensores del conjunto de datos de *sensor-data* y con la localización de los sensores del conjunto de datos *sensor-description*.

La segunda query que necesitaremos será para enviar información al conjunto de datos sobre el estado del tráfico. En esta segunda query tendremos que hacer igual que en la primera y deserializar la información contenida en los diferentes conjuntos de datos. Además, tendremos que hacer uso de un conjunto de datos extra. Este conjunto de datos será el que estará conectado con la base de datos para poder comparar con el histórico de datos y saber si el tráfico de las carreteras es mayor o menor que el habitual.

De esta forma tendríamos preparado nuestro procesamiento de datos en tiempo real vinculado con conjuntos de datos especiales que se actualizan cada 1 o 2 minutos, teniendo de esta forma información actual sobre el estado de las carreteras belgas para poder actuar en consecuencia de lo que los datos nos digan que está ocurriendo.

6.6 Automatizando el proceso de despliegue

El proceso de automatización del despliegue del sistema es una parte muy importante del proyecto. Aunque no forma parte directamente del producto, se trata de una característica que aporta valor, ya que permite realizar el despliegue de nuevas características al sistema pudiéndose añadir al entorno de producción en cuestión de segundos.

Automatizar el despliegue de nuestra aplicación proporciona agilidad y versatilidad al producto, así como la posibilidad de poder tener un sistema de versiones, de forma que en cualquier momento podemos volver a desplegar una versión anterior de nuestro producto.

Para conseguir esto, trabajaremos con los pipelines de *build* y *release* de Azure DevOps para construir soluciones conforme vayamos realizando *Pull-Request* en nuestro repositorio, desde la rama de desarrollo a la rama de producción.

6.6.1 Infraestructura como código. Terraform

El paradigma de la infraestructura como código nace de la necesidad de poder gestionar la arquitectura de sistemas de la información enormes de forma eficaz y eficiente. En sistemas pequeños que cuentan con una simple máquina virtual que no va a tener muchas modificaciones tal vez no sea muy buena idea utilizar este paradigma. Pero cada vez los proyectos son más sofisticados y cuentan con más y más configuraciones para aumentar la eficiencia de los sistemas, la seguridad y otros aspectos, de forma que el despliegue de un producto y sistema que cuenta con múltiples elementos resulta mucho más sencillo y nos facilita un ahorro de tiempo enorme, gracias a la utilización de la infraestructura como código.
En este proyecto hacemos uso de Terraform para aplicar este paradigma. Terraform es una herramienta de código abierto, que, aunque no cuenta con soporte para todos los servicios de Azure y sus características, tiene ya implementados gran parte de ellos. Recibe actualizaciones cada semana gracias a la gran comunidad que tiene detrás y sigue expandiendo su funcionalidad día tras día.

En el repositorio podemos encontrar una carpeta que contendrá nuestros archivos de configuración para el despliegue. La forma en la que vamos a trabajar con Terraform será la siguiente.

Tendremos un archivo de configuración global. Este archivo llamado *architecture.tf* será el punto de entrada de nuestra aplicación para realizar el despliegue. Después contaremos con lo que denominamos como módulos y ambientes.

Un ambiente hace referencia a los diferentes entornos que tendremos en nuestra aplicación o subproductos dentro de un producto grande. En este sistema utilizaremos estos entornos para separar los entornos de desarrollo y producción.

Un módulo hace referencia a un conjunto de recursos de Terraform y Azure diseñado de tal manera que puede ser reutilizado en múltiples proyectos cambiando los parámetros de configuración de los que cuenta. Los módulos son muy útiles para agilizar el desarrollo de los scripts de Terraform. Un módulo bien desarrollado se trata de un caso muy genérico que puede servir para ser aplicado en cualquier contexto dentro de un proyecto de tal forma que solamente tendremos que hacer referencia al módulo y pasarle los parámetros de configuración que necesite.

Por otro lado, cada ambiente tendrá sus propios archivos de estado. En Terraform podemos configurar que se guarde el estado de un despliegue para que cada vez que queramos desplegar solo se desplieguen los cambios en caso de que los haya.

En la figura 59 podemos ver como quedaría la estructura de nuestro directorio de archivos de Terraform.

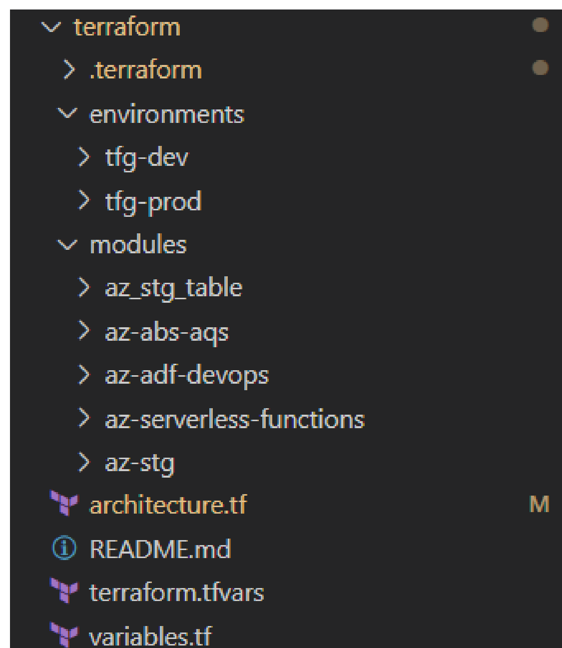


Figura 59. Estructura de archivos de Terraform

.

En el script de despliegue de Terraform contaremos con el despliegue de prácticamente todos los servicios que tiene el sistema, excepto de un par de ellos. Terraform No cuenta con soporte para toda la funcionalidad que necesitamos de Azure Stream Analytics, así que con el primer despliegue del sistema tendremos que desplegar manualmente una de las entradas de datos, la que hace referencia a una tabla de nuestra base de datos, y las dos salidas de datos que están integradas con Power BI.

Por último, tendremos que generar el *token* de autenticación de Databricks para que se pueda acceder a su espacio de trabajo desde otros servicios, para añadirlo al Key Vault y que Azure Data Factory pueda lanzar ejecuciones de los *notebooks* de procesamiento desde allí.

El despliegue de Terraform se llevará a cabo desde un pipeline de Azure DevOps.

6.6.2 DevOps: Pipelines para construir la solución

Ahora toca preparar la parte para automatizar la construcción de nuestra solución cada vez que hagamos cambios en nuestro sistema. Este *pipeline* consistirá en la construcción de nuestra solución cada vez que añadamos nuevas características al producto y hagamos un *pull-request* desde el repositorio de desarrollo al de producción.

Para construir el *pipeline* haremos uso de Azure DevOps y de un archivo de configuración en YAML. Hemos definido los diferentes pasos que se realizan ante una acción. Más concretamente, este *pipeline* se lanzará automáticamente cada vez que hagamos un *pull-request* a la rama de producción.

Las tareas que realizará este pipeline será coger los archivos de nuestro repositorio, y construir con ellos un artefacto con el cual, en un siguiente paso podamos desplegar la solución en Azure. Este paso se conoce como Integración Continua. (ver Figura 60).

De esta forma, el pipeline creará un entorno donde se encontrarán todos los archivos referentes a una misma versión. Juntando esto con el pipeline de despliegue tendremos una forma de poder replicar una versión anterior de nuestro sistema en cualquier momento.

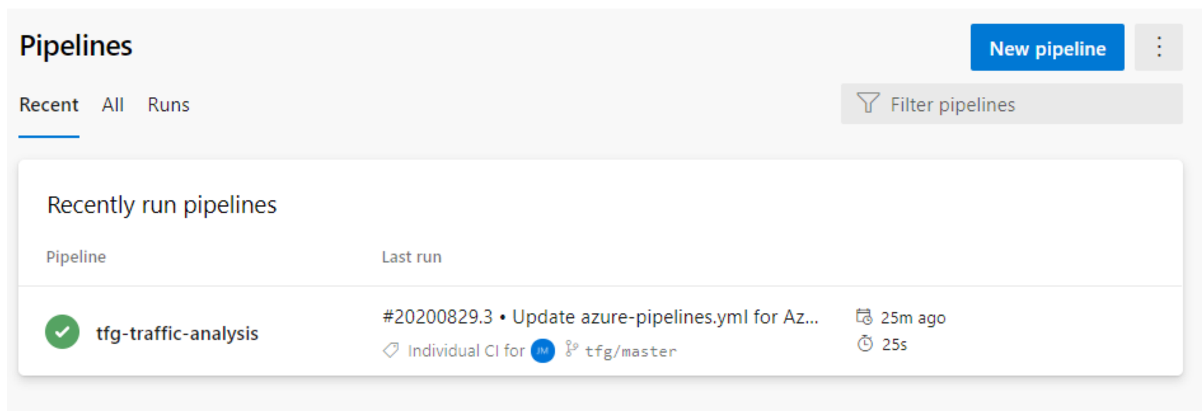


Figura 60. Ejecución del *pipeline* para construir la solución.

6.6.3 DevOps: Pipelines para desplegar la solución

Una vez que tenemos nuestra solución lista para ser completada, el siguiente paso es desplegarla en el entorno de producción. Para ello vamos a utilizar un pipeline de despliegue, que se ejecute cada vez que se genere una nueva versión de nuestro sistema.

De esta forma, cada vez que hagamos un PR a la rama de producción se generará una nueva versión de nuestro producto y al generarse esta nueva versión.

El flujo de trabajo de nuestro *pipeline* de despliegue sería el siguiente:

- Desplegamos la infraestructura haciendo uso de Terraform y el *script* que hemos preparado.
- Desplegamos los *notebooks* de *Databricks.*
- Desplegamos la Azure Function.

Para poder desplegar nuestros recursos haciendo uso de Terraform necesitaremos instalar la extensión de Azure DevOps de Terraform a través del MarketPlace de Azure DevOps. Además, necesitaremos crear un Azure *Service Principal* para realizar el despliegue con él a través de nuestro Pipeline.

Para crear un *Service Principal*, podemos hacerlo de diferentes formas. Se puede hacer a través de Azure CLI (Azure Command Line Interface) o a través del Portal. En la figura 61 se puede ver cómo se puede crear un *Services Principal* a través de Azure CLI de forma muy rápida.

```
az ad sp create-for-rbac --role="Contributor" --scopes="/subscriptions/<your-subscription>"
```

Figura 61. Creando un Service Principal.

Una vez tenemos el *Service Principal* listo, podemos empezar a construir nuestro *pipeline.* El pipeline consta de dos partes. La primera parte se trata de de los artefactos que utilizaremos en el *pipeline.* Estos artefactos son los que se consiguen tras ejecutarse el primer *pipeline*, el *pipeline* que se encarga de construir la solución. La segunda parte es donde se desplegarán nuestros recursos. Desplegamos los *notebooks* de databricks a su workspace correspondiente y las funciones de la Azure Function a su función.

Cabe destacar que, la primera ejecución de nuestro pipeline de despliegue tiene varios pasos, mientras que las siguientes sólo contarán con un paso, el despliegue en sí. En el primer despliegue tendremos que dividir el proceso en 2 partes. La primera parte será para desplegar todos los recursos mediante Terraform. Una vez desplegados los recursos tendremos que generar el token de autenticación en Databricks y subirlo a nuestro Key Vault y añadirlo a las variables de nuestro *pipeline* de DevOps. En la figura 62 podemos ver cómo queda el *pipeline* de despliegue.

Esto es porque hasta que los recursos no están creados no podemos generar las credenciales para poder autenticarnos y desplegarlos los múltiples *notebooks* que componen la solución. De igual manera tendremos que esperar a que esté generado el recurso para obtener la dirección de nuestro entorno de Databricks.

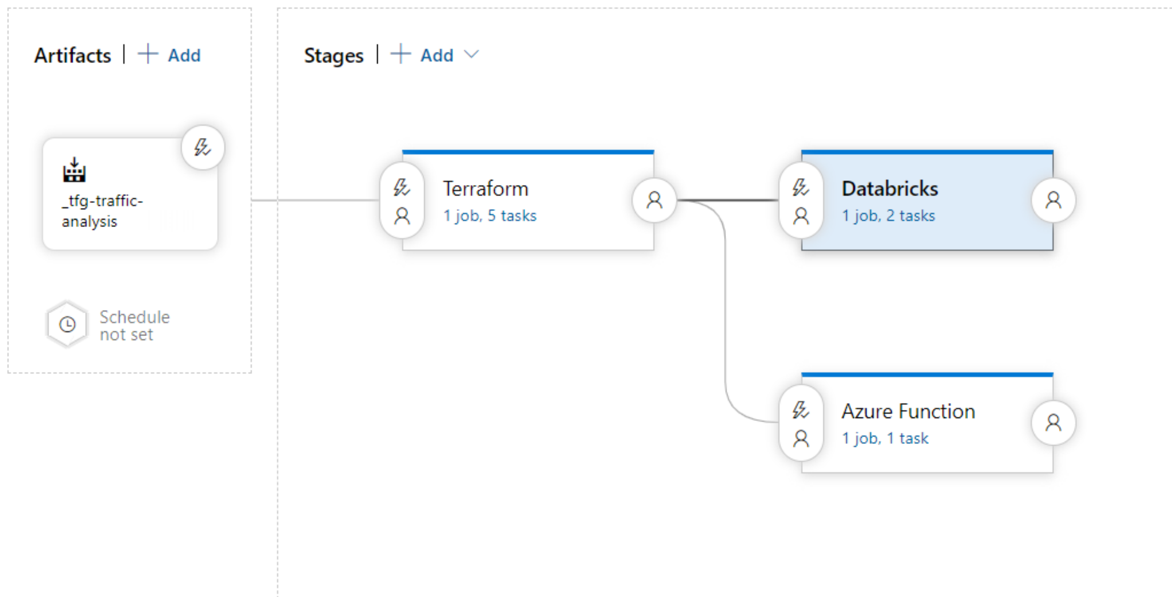


Figura 62. Pipeline de despliegue en DevOps.

7. Resultados. Visualizando los datos procesados en Power BI

Ahora ya tenemos el sistema preparado para procesar todos los datos recibidos y moldeados para poder ser utilizados en nuestros informes. Vamos a comenzar a ver los resultados de nuestros datos, así como los diferentes informes que contiene nuestra aplicación para realizar la monitorización del estado de las carreteras belgas, los sensores e informe de afluencia de gente.

7.1 Informe en tiempo real del estado de los sensores

En este primer informe vamos a poder realizar un informe del estado de los sensores que se encuentran instalados en las carreteras belgas. El informe se ha construido con la herramienta de Power BI, y puede ser visualizado a través de Internet.

En la figura 63 podemos ver como es el informe de datos para poder tener un modelo visual sobre el que comenzar a explicar el modo de empleo y las diferentes áreas del informe.

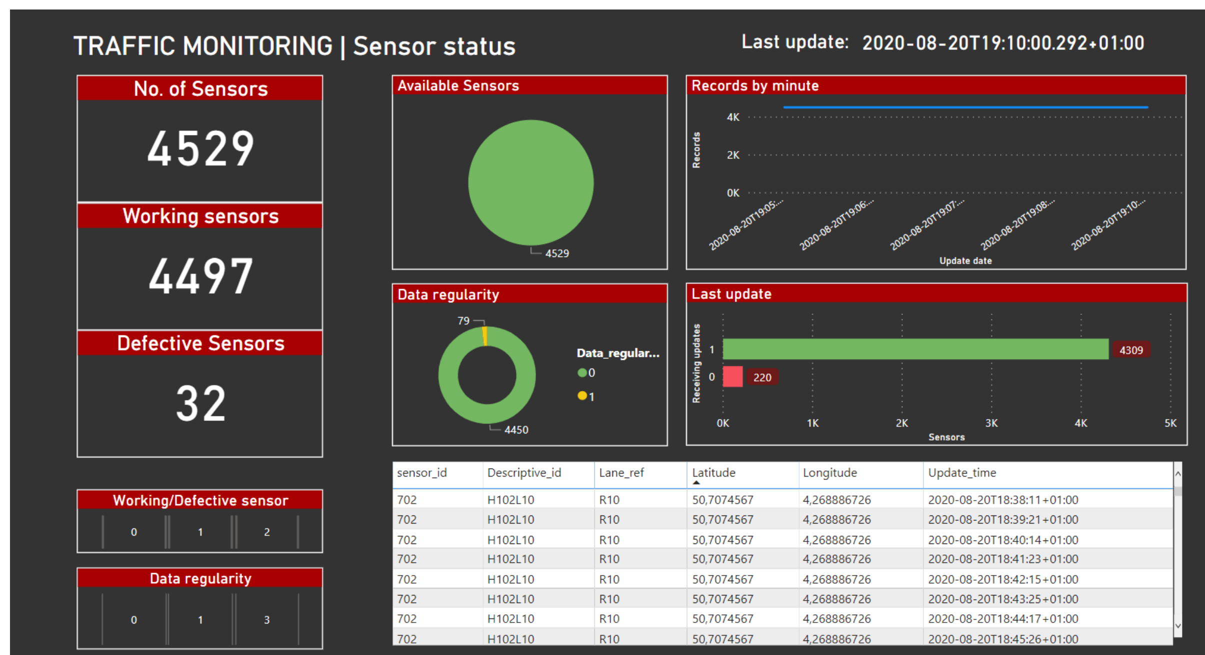


Figura 63. Informe estado de los sensores. Vista general.

El informe está dividido en varias partes. En primer lugar, empezando de izquierda a derecha, podemos encontrarnos en la parte superior y bajando, un recuento de sensores divididos en varias categorías. Más concretamente contamos con un recuento del total de sensores, funcionen o no, y luego un recuento de los que están funcionando perfectamente y de los que tienen algún tipo de problema.

Justo debajo de las celdas que muestran un recuento de los estados generales de los sensores nos encontramos con unas celdas con botones para poder filtrar resultados de los sensores. (ver Figura 64).

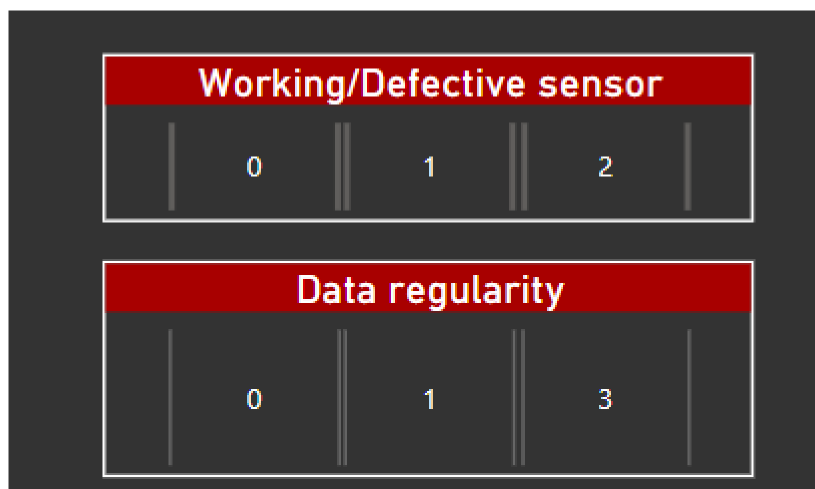


Figura 64. Filtrado de datos en informe de estado de los sensores.

Estos campos serán dinámicos de forma, y cada filtro tiene sus propios valores. Los valores filtrados se encontrarán en la tabla de abajo a la izquierda donde podremos ver los datos relacionados con los sensores filtrados.

En el filtro de "*Working/Defective sensor*" podremos tener valores dentro de un rango de [0-2] siendo cada valor como sigue:

- 0: Sensor sin fallos.
- 1: Sensor con probabilidades de estar fallando. No tiene por qué estar fallando, pero sería bueno su revisión.
- 2: Sensor con fallos severos. Más del 20% de los datos que envía son incorrectos.

Por otro lado, en el filtro de "*Data regularity*" podremos contar con valores dentro de un rango de [0-3] siendo cada valor como sigue:

- 0: Envío de datos de forma regular.
- 1: Envío de datos de forma irregular.
- 2: Envío de datos de forma muy irregular.
- 3: Envío de datos de forma muy irregular debido a fallos.

En la parte derecha del informe contamos con una serie de gráficos, donde podremos ver de forma visual el estado de los sensores, sin necesidad de empezar a filtrar de primera los sensores para ver como están. Entre estos gráficos podemos encontrar un gráfico que nos dirá el número de registros que hemos recibido minuto a minuto. Este gráfico debería ser lineal ya que siempre deberíamos recibir la misma cantidad de datos. Una bajada en el número de registros por minuto podría significar fallos en los sensores, ya sean de funcionamiento, conectividad o cualquier otro fallo (ver Figura 65).

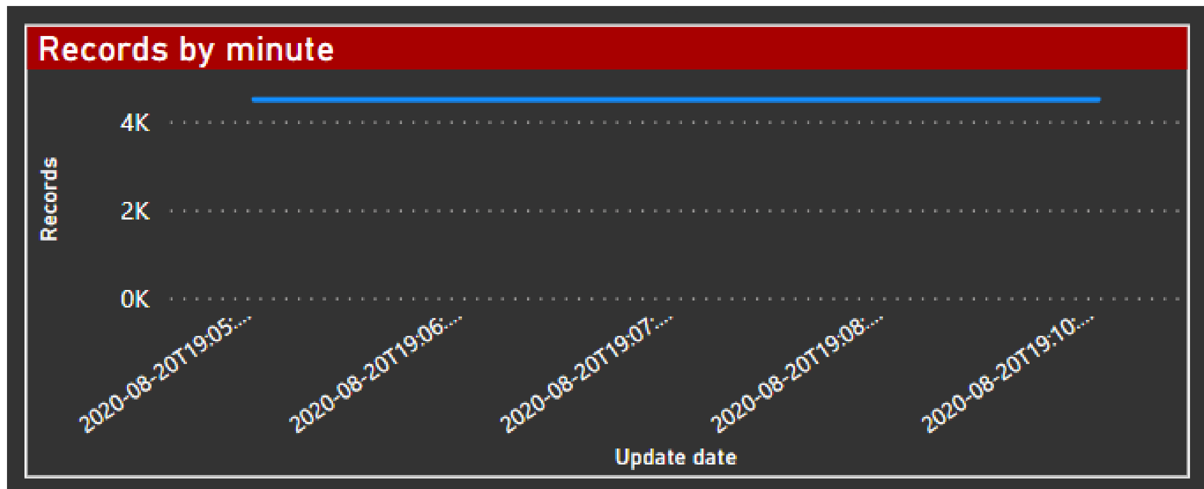


Figura 65. Recuento de sensores minuto a minuto.

Por otro lado, justo debajo del gráfico que nos cuenta el número de registros por minuto, tenemos una gráfica de barras que nos recuenta en número de sensores que están reportando nuevos datos en el último minuto (ver Figura 66).

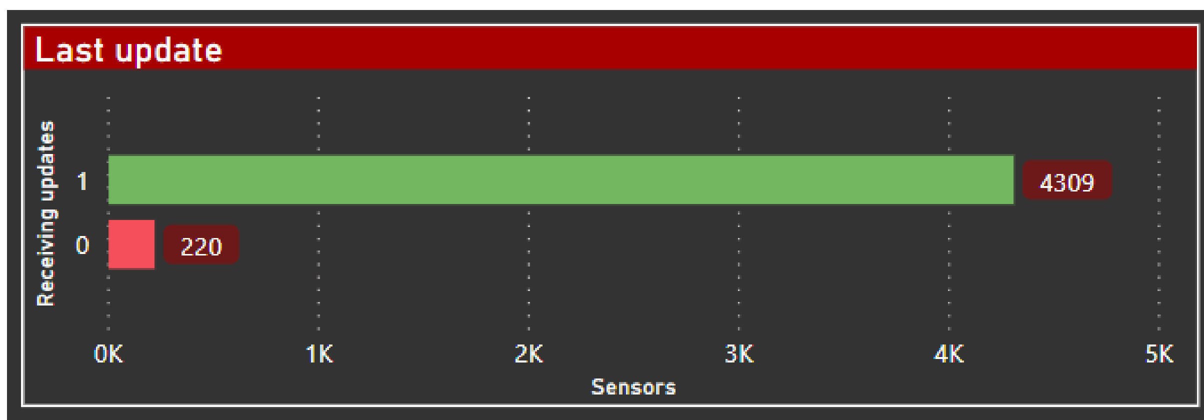


Figura 66. Recuento de sensores que están enviando datos nuevos.

La gráfica que se encuentra a la izquierda de la gráfica de la figura 65, se trata de un gráfico de sectores donde podremos identificar rápidamente si hay algún sensor que presenta fallos de conectividad y que ni siquiera podemos acceder a su estado. En la figura 67 podemos ver que en estos momentos todos los sensores son accesibles y están disponibles, aunque no todos estén funcionando de la mejor forma.

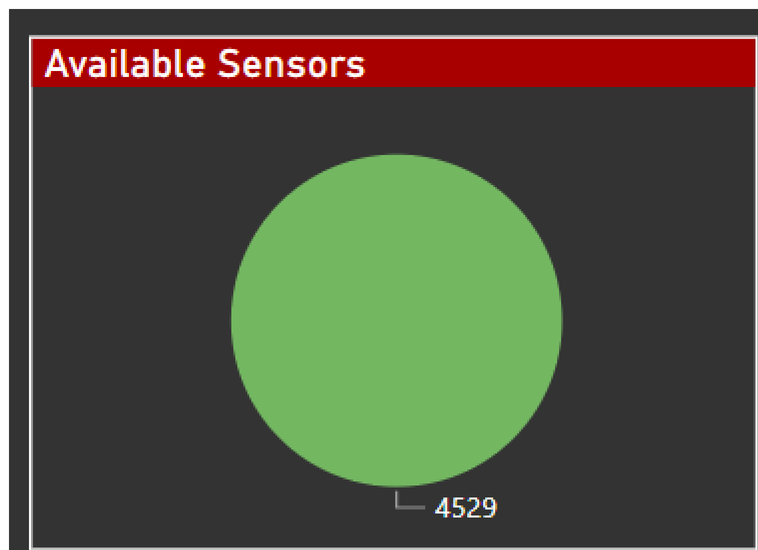


Figura 67. Recuento de sensores que están disponibles.

Por último, contamos con una última gráfica en la cual podremos ver con que regularidad estamos recibiendo datos de cada uno de los sensores. Muestra los mismos datos que obtendremos en la tabla con el filtro de "*Data regularity*". (ver Figura 68)

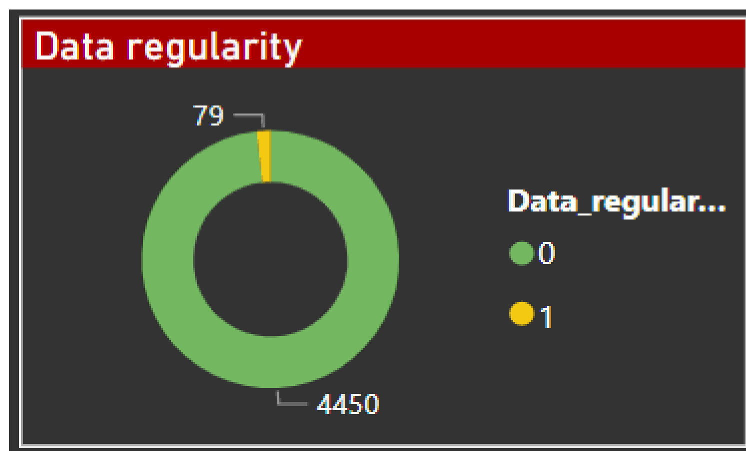


Figura 68. Recuento de sensores agrupados por regularidad de los datos.

Finalmente, como hemos podido ver en la figura 63, este sería el resultado del procesamiento de los datos en tiempo real para conseguir saber el estado de los sensores. El estado de actualización conseguido ronda alrededor de los 10 a 20 segundos en la mayoría de las ejecuciones a excepción de unas pocas ejecuciones que tardan hasta 2 minutos. Por tanto, el resultado conseguido cumple con uno de los requisitos de nuestro sistema que era conseguir saber el estado de los sensores a tiempo real .

7.2 Informe en tiempo real del estado del tráfico

Como segundo requisito de nuestro sistema teníamos la tarea de conseguir un informe de datos a tiempo real que nos permitiese saber el estado del tráfico con la mayor frecuencia de actualización posible que nos permitieran las fuentes de datos, con el objetivo de poder reaccionar ante cambios en la cantidad de tráfico en una vía sin explicación rápida aparente.

Para esta solución, se ha utilizado también un informe construido con la herramienta de Power BI, apoyada de los datos recibidos a través de Stream Analytics. En la figura 69 podemos ver el resultado final del informe.

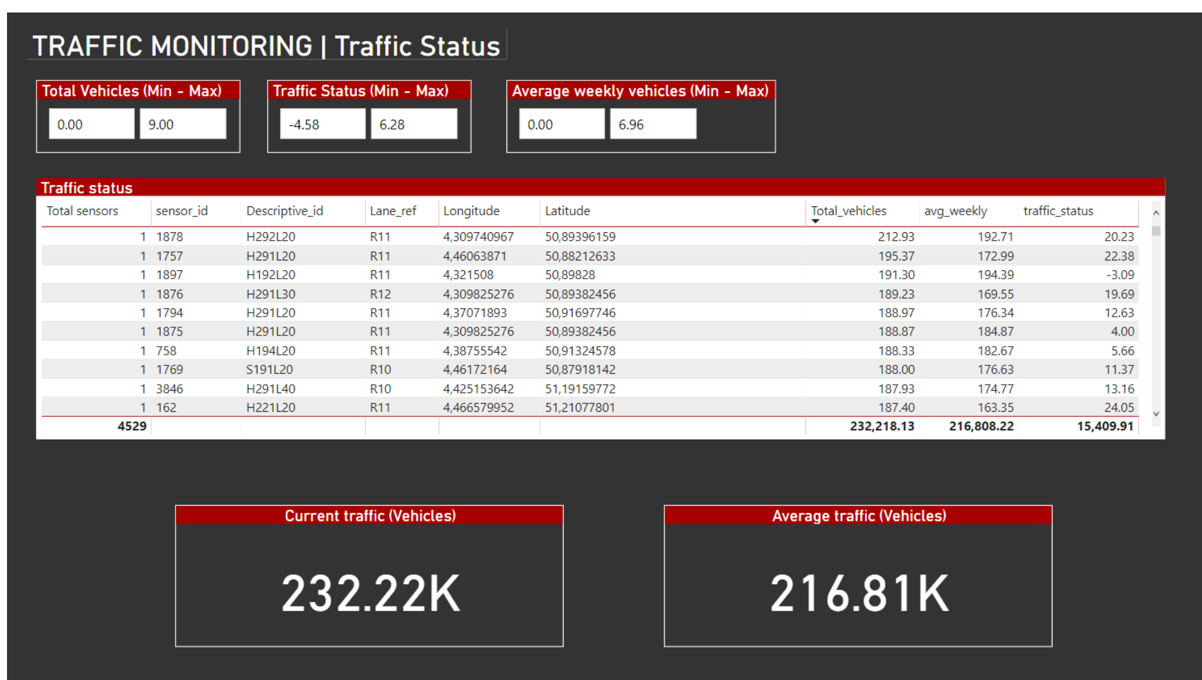


Figura 69. Informe sobre estado del tráfico.

Este informe podemos ver que a simple vista es mucho más simple que el del estado de los sensores. En la parte superior del informe podemos ver como contamos con una serie de filtros para poder ver si hay alguna carretera que tenga una afluencia de vehículos muy superior o muy por debajo de los niveles que queramos configurar.

En la tabla central podremos ver los sensores con los niveles que están marcando, así como la localización y el sensor del que se trata.

Por último, en la parte inferior del informe podemos ver el estado actual total de las carreteras a nivel general, teniendo un recuento de vehículos en todas las carreteras por hora, así como la media que se suele tener.

Finalmente, gracias a los filtros aplicables y a la tabla de resultados, podremos satisfacer el segundo requisito y detectar anomalías en las carreteras belgas en función de la información que los datos nos transmiten.

7.3 Informe de afluencia de personas para vender espacio publicitario

Como tercer y último reporte tenemos el informe que nos muestra los datos de afluencia de gente. Este reporte no necesita ser actualizado a todo momento como los otros 2 informes, por lo que el rango de actualización de este informe será diario y en él se podrán consultar datos históricos de afluencia. Podemos ver como es este informe en la figura 70.

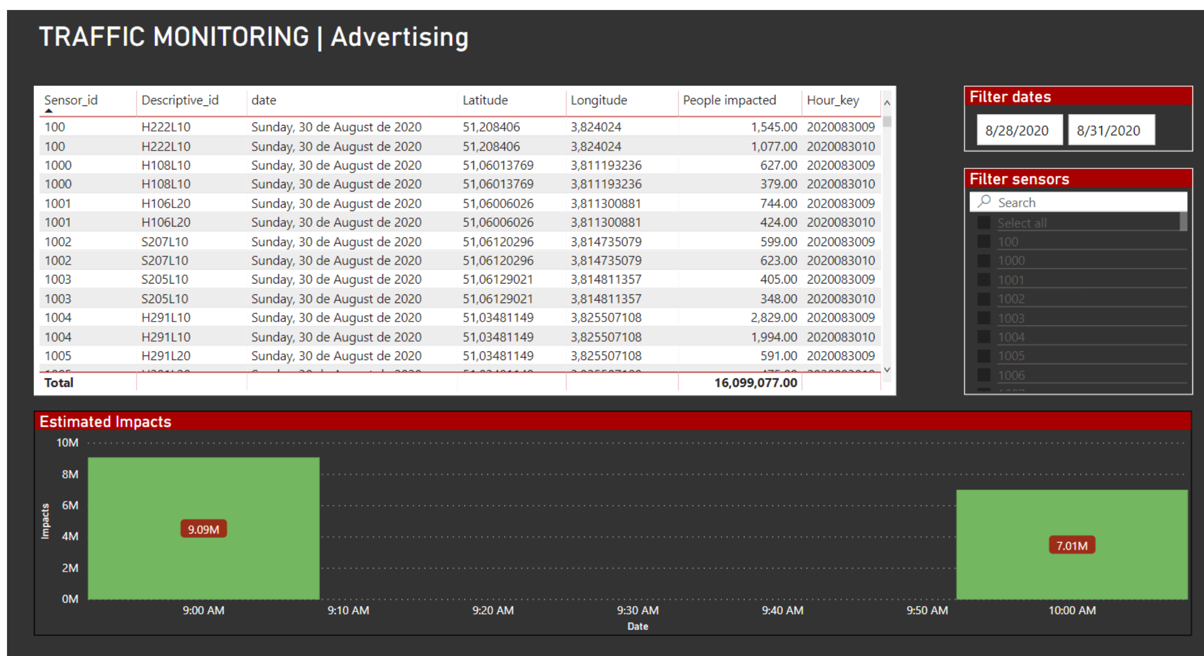


Figura 70. Informe de afluencia de personas por carretera y hora.

En la parte superior del informe podemos observar una tabla en la cual podremos ver los impactos a nivel de hora y tramo de vía. A la derecha de esta misma tabla tendremos un filtro para ajustar las fechas a las que necesitemos ver.

Por último, en la gráfica inferior podremos ver los impactos a nivel de hora, donde podemos ver como varían los impactos y afluencia de personas por las carreteras a nivel general a lo largo del tiempo entre la fecha de inicio marcada y la fecha de fin. Hay que tener en cuenta que estos impactos serán los máximos posibles en todo momento.

De esta forma, con el recuento y exposición de la información que los datos nos dan sobre el número de vehículos y el tipo de vehículos, podemos conseguir este informe donde poder ver a cada hora y a nivel de sensor cuantos impactos ha tenido, consiguiendo esta valiosa información histórica para actuar en consecuencia.

8. Conclusiones y trabajo futuro

8.1 Conclusiones

En un mundo donde cada vez hay más oficios relacionados con los datos, tanto para su tratamiento, procesamiento, análisis o cualquier otra tarea, con este proyecto he querido exponer un ejemplo de caso real y realizar un acercamiento de lo que supone trabajar en la nube con grandes volúmenes de datos y la nube, como punto de entrada para que más personas encuentren el mundo del big data como algo más cercano y sencillo de lo que parece a primera vista.

Durante el proceso de desarrollo de la arquitectura e implementación de sistemas de este tipo te das cuenta de que cuando trabajas con cantidades enormes de datos no puedes capturar todos los posibles escenarios al primer intento o acercamiento de solución.

Cuando estaba desarrollando la solución, sobre todo al principio tuve que cambiar la arquitectura de la solución un par de veces, debido a los costes que suponían procesar y adquirir grandes volúmenes de datos. Esto es algo que cuando entras al mundo real hay que tener en cuenta, ya que se cuentan con unos recursos limitados que hacen que se tengan que tomar decisiones a la hora de implementar la solución.

Tal es así que finalmente se ha conseguido construir un sistema en la nube que procesa más de 6,5 millones de eventos de forma diaria y en tiempo real por un coste en torno a 2 € al día.

Otra experiencia aprendida durante el desarrollo de este proyecto es que, en el mundo del Big Data, un fallo o una mala decisión puede hacer perder mucho dinero a la empresa o equipo que está desarrollando un producto en la nube. El desarrollo en la nube hace que debas tener en cuenta muchos factores como seguridad, no tan solo de los datos, si no al respecto de la administración y manejo de los flujos de datos que se construyen.

De igual forma, he aprendido acerca de muchas tecnologías muy útiles como son Spark y su API de Structured Streaming que junto con Delta Lake permiten tener un mayor control y seguridad sobre los flujos de datos facilitando estas tareas y aportando un gran valor a los productos que las utilizan.

Finalmente, cabe destacar que he aprendido, ya no solo a construir flujos de datos en la nube teniendo que pensar por mí mismo la mejor arquitectura a implementar, sino también la diferencia entre los datos y la información, ya que hay muchas empresas que cuentan con muchísimos datos, pero sin embargo no obtienen ninguna información de esos datos, no pudiendo conseguir información de valor de esa cantidad de datos. Porque un dato por sí solo no te dice nada. Hay que trabajar y extraer la información que ese dato quiere y puede darnos.

8.2 Trabajo futuro

Aunque la plataforma de datos que hemos construido puede por sí sola ya dar una información e idea acerca de los resultados que se buscaban, el sistema aún cuenta con un gran margen de mejora.

Entre los puntos a mejorar se encuentran por ejemplo elaborar un sistema de apoyo a la plataforma de datos para monitorear el funcionamiento de la misma. Para esto Azure cuenta con recursos que ayudan a esto mismo. Por ejemplo, sae podría hacer uso de Azure Log Analytics, Azure Application Insight y los *Dashboards* de Azure para crear un tablero que mostrase la información recogida por los otros dos servicios de monitorización donde podamos ver en todo momento el estado de nuestro sistema y si se encuentra funcionando correctamente.

Otro punto más de mejora es el hecho de realizar un análisis de los datos más en profundidad donde poder extraer información más detallada y poder satisfacer más casos de uso y expandir las aplicaciones de estos datos a otros ámbitos.

Además, se podría añadir una parte de ciencia de datos al producto de forma que por ejemplo se pudiese predecir un acontecimiento antes de que sucedan como puede ser el número de impactos debido a la afluencia a futuro.

En resumen, la plataforma de datos está preparada para trabajar con los requisitos mínimos estipulados, pero si se quisieses se podría expandir y mejorar muchísimo más creando un sistema mucho más sofisticado.

Bibliografía

[1] Belgian Open Data Portal. Listado de sensores. Llamada a la API. Recuperado 1 de septiembre de 2020, de http://miv.opendata.belfla.be/miv/configuratie/xml
[2] Belgian Open Data Portal. Documentación. Listado de sensores. Recuperado 1 de septiembre de 2020, de http://miv.opendata.belfla.be/miv-config.xsd
[3] Belgian Open Data Portal. Mediciones de los sensores. Llamada a la API. Recuperado 1 de septiembre de 2020, de http://miv.opendata.belfla.be/miv/verkeersdata
[4] Belgian Open Data Portal. Documentación. Mediciones de los sensores. Recuperado 1 de septiembre de 2020, de http://miv.opendata.belfla.be/miv-verkeersdata.xsd
[5] Microsoft Azure. Recuperado 1 de septiembre de 2020, de https://azure.microsoft.com/es-es/
[6] Documentación. Azure Data Factory. Recuperado 1 de septiembre de 2020, de https://docs.microsoft.com/es-es/azure/data-factory/
[7] Documentación. Azure Function App. Recuperado 1 de septiembre de 2020, de https://docs.microsoft.com/es-es/azure/azure-functions/
[8] Documentación. Azure Logic App. Recuperado 1 de septiembre de 2020, de https://docs.microsoft.com/es-es/azure/logic-apps/
[9] Documentación. Azure Storage Account. Recuperado 1 de septiembre de 2020, de https://docs.microsoft.com/es-es/azure/storage/
[10] Bill Inmon (2016). Data Lake Architecture: Designing the Data Lake and Avoiding the Garbage Dump. Technics Publications
[11] Documentación. Azure Table Storage. Recuperado 1 de septiembre de 2020, de https://docs.microsoft.com/es-es/azure/storage/tables/
[12] Documentación. Stream Analytics. Recuperado 1 de septiembre de 2020, de https://docs.microsoft.com/es-es/azure/stream-analytics/
[13] Documentación. Power BI. Recuperado 1 de septiembre de 2020, de https://docs.microsoft.com/es-es/power-bi/
[14] Documentation. Tumbling Window. Recuperado 1 de septiembre de 2020, de https://docs.microsoft.com/es-es/azure/data-factory/how-to-create-tumbling-window-trigger
[15] Documentación. Azure Event Grid. Recuperado 1 de septiembre de 2020, de https://docs.microsoft.com/es-es/azure/event-grid/
[16] Documentación. Azure Storage queue. Recuperado 1 de septiembre de 2020, de https://docs.microsoft.com/es-es/azure/storage/queues/
[17] Documentación. Azure Databricks. Recuperado 1 de septiembre de 2020, de https://docs.microsoft.com/es-es/azure/databricks/
[18] Documentación. Spark y Pysaprk. Recuperado 1 de septiembre de 2020, de https://spark.apache.org/docs/latest/
[19] Documentación. Azure SQL Database. Recuperado 1 de septiembre de 2020, de https://docs.microsoft.com/es-es/azure/azure-sql/
[20] Documentación. Python. Recuperado 1 de septiembre de 2020, de https://docs.python.org/es/3.8/tutorial/index.html
[21] Wes McKinney (2017). Python for Data Analysis, 2nd Edition. O'Reilly Media
[22] Documentación. Delta Lake. Recuperado 1 de septiembre de 2020, de https://docs.delta.io/latest/index.html

[23] Documentación. Structured Streaming. Recuperado 1 de septiembre de 2020, de https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html
[24] Documentación. T-SQL. Recuperado 1 de septiembre de 2020, de https://docs.microsoft.com/es-es/sql/t-sql/language-reference?view=sql-server-ver15
[25] Documentación. Terraform. Recuperado 1 de septiembre de 2020, de https://www.terraform.io/intro/index.html
[26] Documentación. Parquet. Recuperado 1 de septiembre de 2020, de https://parquet.apache.org/documentation/latest/
[27] Documentación. Azure DevOps. Recuperado 1 de septiembre de 2020, de https://docs.microsoft.com/en-us/azure/devops/?view=azure-devops
[28] Documentación. Azure SQL Database. Recuperado 1 de septiembre de 2020, de https://docs.microsoft.com/en-us/azure/azure-sql/database/sql-database-paas-overview
[29] Documentación. Azure Key Vault. Recuperado 1 de septiembre de 2020, de https://docs.microsoft.com/es-es/azure/key-vault/
[30] Documentación. Azure Event Hub. Recuperado 1 de septiembre de 2020, de https://docs.microsoft.com/es-es/azure/event-hubs/
[31] Documentación. Azure Durable Functions. Recuperado 1 de septiembre de 2020, de https://docs.microsoft.com/es-es/azure/azure-functions/durable/durable-functions-overview
[32] Documentación. Azure IoT Edge. Recuperado 1 de septiembre de 2020, de https://azure.microsoft.com/es-es/services/iot-edge/
[33] Documentación 1. Arquitectura Delta. Recuperado 1 de septiembre de 2020, de https://www.youtube.com/watch?v=FePv0lro0z8
[34] Documentación 2. Arquitectura Delta. Recuperado 1 de septiembre de 2020, de https://databricks.com/wp-content/uploads/2019/01/Databricks-Delta-Technical-Guide.pdf
[35] Documentación. Arquitectura Lambda. Recuperado 1 de septiembre de 2020, de https://docs.microsoft.com/es-es/azure/architecture/data-guide/big-data/
[36] Documentación 2. Arquitectura Lambda. Recuperado 1 de septiembre de 2020, de https://databricks.com/glossary/lambda-architecture
[37] Documentación. Arquitectura Kappa. Recuperado 1 de septiembre de 2020, de https://docs.microsoft.com/es-es/azure/architecture/data-guide/big-data/#kappa-architecture
[38] Christopher Adamson (2010). Star Schema The Complete Reference. Mc Graw hill.
[39] Databricks. The Data Engineer's Guide to Spark and Delta Lake. Recuperado 1 de septiembre de 2020, de https://databricks.com/p/ebook/data-engineer-spark-guide
[40] Databricks. 8 Steps for a developer to learn apache spark with Delta lake. Recuperado 1 de septiembre de 2020, de https://databricks.com/p/ebook/learn-apache-spark-with-delta-lake
[41] Parquet. Columnar storage pictures: Recuperado 1 de septiembre de 2020, de https://community.ptc.com/t5/IoT-Tech-Tips/Parquet-Data-Format-used-in-ThingWorx-Analytics/td-p/535228
[42] Bill Chambers, Matei Zaharia (2018). Spark: The Definitive Guide. O'Reilly Media.
[43] Yevgeniy Brikman (2019) Terraform: Up & Running, 2nd Edition. O'Reilly Media.
[44] Anindita Basak, Krishna Venkataraman, Ryan Murphy, Manpreet Singh (2017). Stream Analytics with Microsoft Azure. Packt Publishing.
[45] Anuj Kumar (2018). Architecting Data-Intensive Applications. Packt Publishing.

UNIVERSIDAD DE ALMERÍA

Resumen/Abstract

A día de hoy el Big Data y la nube están en la mayoría de servicios que utilizamos sin que ni siquiera seamos conscientes. Son un una necesidad para el continuo desarrollo a día de hoy.

Traffic Monitoring se trata de una plataforma de datos con la cual se procesan los datos que provienen de sensores de tráfico rodado que el gobierno belga tienen instalados a lo largo de sus carreteras.

El sistema es capaz de procesar millones de registros por hora a un coste muy bajo a la vez que es capaz de servir resultados e información relevante en un periodo de tiempo muy bajo manteniendo una latencia aceptable a la hora de ofrecer la información en tiempo real.

El producto ha sido desarrollado utilizando tecnologías que están siendo utilizadas por las compañías más punteras del mundo en el momento de su desarrollo como son BP, Alibaba, Tencent Games , McAfee o UpWork como pueden ser Delta Lake o Spark.

Nowadays the Big Data and Cloud are in most of the services we use and we aren’t aware of it . They are a necessity for our continuous development in the world today.

Traffic Monitoring is a data platform built in Azure that processes data retrieved from traffic sensors installed in the Belgium roads by the government.

The system is capable of processing millions of records per hour at a very low cost while at the same time it is able to serve relevant results and information in a very low period of time while maintaining an acceptable latency in providing the information in real time.

The product has been developed using technologies that are being used by the world’s leading companies such as BP, Alibaba, Tencent Games , McAfee or UpWork Some of the technologies are Delta Lake and Spark.

GRADO EN INGENIERÍA INFORMÁTICA, 2019/2020