1

INGENIERÍA DE DATOS

1.1 INTRODUCCIÓN

La ingeniería de datos es un campo fundamental en la ciencia de datos y la analítica moderna que se encarga de gestionar y preparar los datos para su análisis. En un mundo donde la cantidad de datos generados diariamente crece exponencialmente, la ingeniería de datos desempeña un papel crucial en la obtención de información valiosa a partir de esta avalancha de información. Se trata de un proceso que abarca desde la recolección y almacenamiento de datos hasta su limpieza, transformación y análisis.

La principal meta de la ingeniería de datos es proporcionar a los profesionales de datos, científicos de datos e ingenieros de aprendizaje automático, un conjunto de datos de alta calidad y bien estructurado que sea adecuado para realizar análisis y entrenar modelos de aprendizaje automático. Esto implica diversas etapas y técnicas que incluyen:

- **Captura de datos**: en esta fase, se recopilan datos de múltiples fuentes, como bases de datos, archivos de registro, sensores, redes sociales y más. La captura debe ser planificada y organizada para asegurarse de que se recojan los datos relevantes.
- **Almacenamiento**: los datos se almacenan en sistemas de gestión de bases de datos o en plataformas de almacenamiento distribuido como Hadoop o AWS S3. La elección del sistema de almacenamiento depende de factores como el volumen de datos y los requisitos de rendimiento.
- **Limpieza de datos**: los datos a menudo contienen errores, valores atípicos y datos faltantes. La limpieza de datos implica la identificación y corrección de estos problemas para garantizar la integridad de los datos.

- **Transformación**: los datos se transforman para que se ajusten a las necesidades del análisis. Esto puede incluir la agregación de datos, la creación de nuevas características y la normalización de valores.
- **Integración**: en proyectos de ingeniería de datos, es común trabajar con datos de diversas fuentes. La integración implica combinar estos datos de manera coherente para obtener un conjunto de datos unificado.
- **Automatización**: para proyectos a gran escala, es importante automatizar muchas de las tareas de ingeniería de datos para garantizar la eficiencia y la coherencia en el procesamiento de datos.
- **Seguridad y cumplimiento**: la seguridad de los datos y el cumplimiento de las regulaciones son aspectos críticos en la ingeniería de datos. Se deben implementar medidas de seguridad para proteger la privacidad y la integridad de los datos.
- **Escalabilidad**: los sistemas de ingeniería de datos deben ser escalables para manejar grandes volúmenes de datos a medida que la organización crece.

En resumen, la ingeniería de datos es un proceso esencial para aprovechar el potencial de los datos en la toma de decisiones empresariales, la investigación y el desarrollo de modelos de aprendizaje automático. Es el primer paso hacia la obtención de información significativa y valiosa a partir de la gran cantidad de datos disponibles actualmente.

1.2 IMPORTANCIA DE LOS DATOS EN LAS ORGANIZACIONES

Los datos permiten tomar decisiones de una manera informada y basada en evidencias. Por ejemplo, cuando vamos a salir de casa, consultamos el pronóstico del clima para decidir qué ropa nos vamos a poner. A partir de los datos que nos provee el pronóstico del clima, podemos tomar una decisión "data-driven" o dirigida por los datos.

En un ámbito profesional, cuando una organización analiza sus datos, puede descubrir patrones y tendencias e inferencias. De esa forma, podrá mejorar sus procesos o brindar mejores servicios a sus clientes. Por ejemplo, Netflix a partir de datos que recolecta de sus usuarios, permite personalizar el contenido que les muestra en su plataforma de acuerdo con diferentes aspectos como gustos, entorno, horario, etc. Incluso, Netflix personaliza la portada de las series y películas que le recomienda a sus usuarios a partir de dichos datos.

En nuestro día a día, usamos dispositivos y aplicaciones que crean y recopilan una cantidad de datos cada vez mayor. Sumado a eso, existe una variedad de métodos y técnicas que permiten, a partir de esos datos crudos, obtener valor e información accionable para tomar decisiones. De eso se trata la analítica de datos, abarcando la recopilación, transformación y organización de datos con el fin de extraer conclusiones, hacer predicciones e impulsar la toma de decisiones fundamentadas en evidencia.

La analítica de datos es muy importante para las organizaciones hoy en día porque ayuda a obtener más visibilidad y un conocimiento más profundo de sus procesos y servicios. De esa forma, pueden identificar oportunidades de mejora y optimización, lo que los llevará a reducir costes y desarrollar mejores productos y servicios centrados en el cliente. Existen cuatro tipos principales de analíticas de datos:

- **Descriptiva**: permite comprender que ocurrió o que está ocurriendo en el entorno de datos. Suele realizarse con visualizaciones como histogramas de los datos.
- **Diagnóstica**: permite entender por qué sucedió algo, por medio de técnicas de minería de datos y de operaciones y transformaciones sobre los datos.
- **Predictiva**: podríamos predecir lo que probablemente ocurrirá en el futuro a partir de los datos históricos por medio de técnicas de machine learning y modelado predictivo.
- **Prescriptiva:** no solo predice lo que puede ocurrir, sino que sugiere una respuesta óptima a ese resultado, ofreciendo el mejor curso de acción para solucionar el problema. Se basa en el uso de análisis de grafos, redes neuronales y motores de recomendación.

1.3 CICLO DE VIDA DE LOS DATOS

El ciclo de vida de los datos se refiere a las diferentes etapas por las que pasan los datos desde su creación hasta su eliminación o archivado. Estas etapas son fundamentales para gestionar los datos de manera eficiente y garantizar su calidad y seguridad a lo largo del tiempo. A continuación, se describen las etapas típicas en el ciclo de vida de los datos:

- Ingesta y almacenamiento.
- Procesamiento y preparación.
- Exploración.
- Experimentación y predicción.

Figura 1.1. Etapas del ciclo de vida de los datos

El ciclo de vida de los datos es una parte crucial de la gestión de datos en una organización, y su implementación efectiva garantiza que los datos sean un activo valioso y seguro a lo largo del tiempo. La gestión adecuada del ciclo de vida de los datos también puede ayudar a optimizar costos de almacenamiento y mejorar la eficiencia operativa.

1.3.1 Ingesta y almacenamiento

Los datos de las organizaciones pueden estar dispersos en diferentes sistemas, bases de datos y repositorios. El primer paso es recolectar los datos de forma periódica y centralizar los mismos en un repositorio o base de datos. De eso se encarga la etapa de ingesta y almacenamiento.

La ingesta es el proceso responsable de la obtención de los datos de diferentes sistemas y orígenes para depositarlos en algún sistema de almacenamiento, que puede ser un repositorio de archivo o un sistema de base de datos. El principal objetivo es adquirir los datos y almacenarlos en algún lugar para permitir la limpieza y exploración de los datos.

1.3.2 Procesamiento y preparación

Una vez recolectados los datos, la siguiente fase es el procesamiento, que toma los datos crudos, los limpia y los convierte en un formato más significativo. El resultado es un conjunto de datos limpio y organizado que se podría utilizar para realizar consultas y generar visualizaciones, dándole la forma y el contexto necesarios para interpretarse.

Algunas de las tareas que se suelen realizan en esta fase son la eliminación de registros duplicados, nulos o erróneos, conversión de columnas de un tipo de datos a otro, creación de nuevas columnas a partir de cálculos, unión de tablas para complementar información. El objetivo es obtener datos lo suficientemente coherentes para que el equipo de análisis y ciencia de datos pueda explorarlos y realizar distintos tipos de analítica.

1.3.3 Exploración

Una vez recolectados los datos, la siguiente fase es el procesamiento, que toma los datos crudos, los limpia y los convierte en un formato más significativo. El resultado es un conjunto de datos limpio y organizado que se podría utilizar para realizar consultas y generar visualizaciones, dándole la forma y el contexto necesarios para interpretarse.

1.3.4 Experimentación y predicción

Una vez que se analizaron los datos históricos, se obtuvieron conclusiones y se descubrieron tendencias, el siguiente nivel es la analítica predictiva. En esta fase, entra en juego la aplicación de técnicas de machine learning sobre los datos. Con los datos disponibles, se realizan experimentos a modo de entrenamiento de los datos para ver cuál es el mejor modelo o algoritmo para realizar predicciones, utilizando diferentes métricas como el accuracy o rendimiento del modelo.

1.4 ROL DE INGENIERO/A DE DATOS

La ingeniería de datos se enfoca en las primeras dos etapas del flujo de trabajo de Data Analytics, la de ingesta y almacenamiento y la de procesamiento y preparación de los datos. Por ello, un ingeniero de datos tiene un rol fundamental y de gran importancia ya que establecen las bases para permitir distintos tipos de analítica.

En la mayoría de las organizaciones, un ingeniero de datos es el responsable principal de integrar, transformar y consolidar datos de varios sistemas de datos estructurados y no estructurados en estructuras adecuadas para crear soluciones de análisis. De esta forma, los ingenieros de datos son responsables de **ingestar y almacenar datos de diferentes fuentes,** de distintos formatos y de distintas estructuras, para que sean fácilmente **accesibles y listos para ser analizados y explotados.**

Además, un ingeniero de datos colabora con otros perfiles como los analistas de datos y científicos para asegurarse de que los datos estén disponibles y sean accesibles para el análisis y la toma de decisiones. Los ingenieros de datos se encargan de construir y mantener las estructuras de datos para que la empresa pueda acceder a ellos fácilmente. Por eso sus tareas pueden incluir diseñar y crear data pipelines.

Un data pipeline es el proceso que se crea para extraer datos de una fuente, transformarla y llevarla a un destino. La fuente puede ser por ejemplo una base de datos o una aplicación en la nube. El destino, normalmente, será un **Data Warehouse** o un **Data Lake.** Para ello**,** los ingenieros de datos deben tener competencias con una variedad de herramientas y lenguajes de scripting, en particular SQL y Python, e idealmente con otros también.

- **SQL**: uno de los ingenieros de datos de lenguajes más comunes que usan es SQL o Lenguaje de consulta estructurado, que es un lenguaje relativamente fácil de aprender. SQL usa consultas que incluyen instrucciones SELECT, INSERT, UPDATE y DELETE para trabajar directamente con los datos almacenados en tablas.
- **Python**: Python es uno de los lenguajes de programación más populares y con un crecimiento más rápido del mundo. Se utiliza para todo tipo de tareas, incluida la programación web y el análisis de datos. En los últimos años ha sido el lenguaje que más ha crecido para su uso en la implementación de modelos de aprendizaje automático y redes neuronales.
- **Otros**: según las necesidades de la organización y el conjunto de aptitudes individuales, también puede usar otros lenguajes como R, Java, Scala, .NET.

El rol de un ingeniero o ingeniera de datos es esencial en el campo de la ciencia de datos y la gestión de datos en las organizaciones. Su trabajo se centra en la adquisición, limpieza, transformación, almacenamiento y disponibilidad de datos para su posterior análisis. A continuación, se describen las responsabilidades y funciones clave de un ingeniero o ingeniera de datos:

- **Adquisición de datos**: recolectar datos de diversas fuentes, como bases de datos, sistemas de registro, aplicaciones web, sensores y archivos externos. Esto implica diseñar y mantener procesos de extracción de datos (ETL) para obtener datos de manera regular y confiable.
- **Almacenamiento de datos**: diseñar y administrar sistemas de almacenamiento de datos que sean eficientes y escalables. Esto puede incluir bases de datos relacionales, almacenes de datos, sistemas de archivos distribuidos y soluciones en la nube como AWS S3 o Azure Data Lake.
- **Limpieza y preprocesamiento de datos**: identificar y abordar problemas en los datos, como valores atípicos, datos faltantes o errores. Esto garantiza que los datos estén limpios y listos para su análisis.
- **Transformación de datos**: aplicar transformaciones a los datos según las necesidades del análisis o de modelización. Esto puede incluir la agregación de datos, la creación de nuevas características y la normalización de los datos.
- **Integración de datos**: combina datos de múltiples fuentes para crear un conjunto de datos coherente y completo. Esto es crucial cuando se trabaja con datos dispersos en diferentes sistemas y ubicaciones.
- **Automatización**: automatizar tareas repetitivas y programar procesos ETL para garantizar la eficiencia y la consistencia en la adquisición y procesamiento de datos.
- **Seguridad de datos**: implementar medidas de seguridad para proteger la confidencialidad, integridad y disponibilidad de los datos. Esto incluye el acceso controlado a los datos y la encriptación cuando sea necesario.
- **Cumplimiento regulatorio**: asegurarse de que los procesos de gestión de datos cumplan con las regulaciones y estándares aplicables, como el Reglamento General de Protección de Datos (GDPR) en la Unión Europea.
- **Monitorización y mantenimiento**: supervisar constantemente la integridad de los datos y el rendimiento de los sistemas de almacenamiento. Realizar mantenimiento preventivo y corrección de problemas.
- **Colaboración**: colaborar con otros profesionales de datos, como científicos de datos y analistas, para comprender las necesidades de los proyectos y proporcionar los datos adecuados.
- **Documentación**: documentar procesos, flujos de datos y esquemas de datos para facilitar la comprensión y el uso de los datos por parte de otros miembros del equipo.
- **Escalabilidad**: diseñar sistemas y procesos que puedan manejar volúmenes crecientes de datos a medida que la organización crece.

El rol de ingeniero o ingeniera de datos es esencial para asegurar que los datos estén disponibles, limpios y listos para el análisis, lo que permite a las organizaciones tomar decisiones informadas basadas en datos y desarrollar modelos de aprendizaje automático eficaces. Además, juegan un papel clave en la seguridad y el cumplimiento de datos, lo que es fundamental en un entorno donde la privacidad y la seguridad de los datos son prioritarios.

1.5 INGESTA DE DATOS

Una de las funciones de la ingeniería de datos es la ingesta, o recolección, de datos. Como sabemos, las organizaciones generan y capturan datos por medio de sus sistemas y aplicaciones. El/la ingeniero/a de datos debe extraer datos de los diferentes sistemas y fuentes de las organizaciones.

Dichas fuentes son muy variadas, van desde fuentes de datos estructuradas hasta fuentes no estructuradas. Nos podemos encontrar con bases de datos relacionales, no relacionales, APIs, repositorios con archivos, ya sea archivos CSV, Excel, imágenes, PDFs, etc.

La ingesta extrae los datos desde la fuente donde se crean o almacenan originalmente y los carga en un destino o zona temporal. Un pipeline de datos sencillo puede que tenga que aplicar una o más transformaciones ligeras para enriquecer o filtrar los datos antes de escribirlos en un destino, almacén de datos o cola de mensajería. Las **fuentes** más comunes desde las que se obtienen los datos suelen ser:

- Servicios de mensajería como Apache Kafka, los cuales han obtenido datos desde fuentes externas, como pueden ser dispositivos IOT.
- Bases de datos relacionales, las cuales se acceden, por ejemplo, mediante JDBC.
- Servicios REST que devuelven los datos en formato JSON.
- Servicios de almacenamiento distribuido como HDFS o S3.

Los **destinos** donde se almacenan los datos son:

- Servicios de mensajería como Apache Kafka.
- Bases de datos relacionales como MySQL.
- Bases de datos NoSQL como MongoDB.
- Servicios de almacenamiento distribuido como HDFS o S3.
- Plataformas de datos como Snowflake o Databricks.

1.5.1 Batch vs streaming

La ingesta de datos es el proceso de recopilar datos de diversas fuentes y llevarlos a un sistema central para su posterior procesamiento y análisis. Hay dos enfoques principales

para la ingesta de datos: **batch** (por lotes) y **streaming** (transmisión en tiempo real). Cada uno tiene sus propias características y casos de uso específicos. A continuación, analizamos las diferencias entre ambos:

- **Ingesta de Datos por Lotes (Batch):** el proceso se ejecuta de forma periódica (normalmente en intervalos fijos) a partir de unos datos estáticos. Muy eficiente para grandes volúmenes de datos, y donde la latencia (del orden de minutos) no es el factor más importante.
  - **Procesamiento por lotes**: en este enfoque, los datos se recopilan, almacenan y procesan en bloques o lotes. Los datos se agrupan en intervalos de tiempo predefinidos y se procesan en lotes completos. Por ejemplo, se pueden recopilar datos cada hora y procesarlos juntos.
  - **Retraso inherente**: debido a la naturaleza por lotes, existe un retraso inherente en el procesamiento de datos. Los datos se recopilan y procesan en intervalos de tiempo, por lo que no se pueden analizar en tiempo real.
  - **Adecuado para grandes volúmenes**: la ingesta por lotes es eficiente cuando se trata de grandes volúmenes de datos que no necesitan analizarse en tiempo real. Se usa en el procesamiento de informes históricos y análisis de tendencias.
  - **Ejemplo de herramienta: A**pache **Hadoop** es una plataforma popular para el procesamiento por lotes que permite el procesamiento eficiente de grandes conjuntos de datos.
- **Ingesta de Datos en Tiempo Real (Streaming):** también conocido como en tiempo real, donde los datos se leen, modifican y cargan tan pronto como llegan a la capa de ingesta (la latencia es crítica). Algunas de las herramientas más utilizadas son Apache Storm, Spark Streaming, Apache Nifi, Apache Kafka.
  - **Procesamiento en tiempo real**: en este enfoque, los datos se transmiten y procesan a medida que se generan, sin retrasos significativos. Esto permite analizar datos en tiempo real y tomar decisiones basadas en información actualizada.
  - **Bajo retraso**: la ingesta en tiempo real minimiza el retraso entre la generación de datos y su disponibilidad para el análisis. Esto es crítico en aplicaciones donde la toma de decisiones instantáneas es esencial, como la detección de fraudes o la monitorización de sistemas.
  - **Adecuado para eventos en tiempo real**: se utiliza cuando es importante detectar patrones o eventos en tiempo real, como transacciones bancarias, seguimiento de usuarios en sitios web o sensores IoT.
  - **Mayor complejidad**: los sistemas de procesamiento en tiempo real pueden ser más complejos de implementar y gestionar debido a la necesidad de manejar flujos continuos de datos y garantizar la tolerancia a fallos.
  - **Ejemplo de herramienta**: Apache **Kafka** es una plataforma de transmisión en tiempo real ampliamente utilizada para la ingesta y el procesamiento de datos en tiempo real.

La elección entre ingesta por lotes y en tiempo real depende de los requisitos específicos del proyecto. Los sistemas de procesamiento por lotes son adecuados para análisis retrospectivos y volúmenes masivos de datos, mientras que la ingesta en tiempo real es esencial cuando se requiere análisis en tiempo real y la toma de decisiones instantáneas. En muchos casos, las organizaciones utilizan una combinación de ambos enfoques para abordar una variedad de necesidades de datos.

1.5.2 Arquitectura de la capa de ingesta

Si nos basamos en la arquitectura por capas, podemos ver como la capa de ingesta es la primera de las capas, la cual recoge los datos que provienen de fuentes diversas. Los datos se categorizan y priorizan, facilitando el flujo de éstos en posteriores capas.

El primer paso de la ingesta es el paso más pesado, por tiempo y cantidad de recursos necesarios. Es normal realizar la ingesta de flujos de datos desde diferentes fuentes de datos, los cuales se obtienen a velocidades variables y en diferentes formatos. Los cuatro parámetros en los que debemos centrar nuestros esfuerzos son:

- **Velocidad de los datos:** cómo fluyen los datos entre las diferentes máquinas cliente y servidores, si el flujo es continuo o masivo.
- **Tamaño de los datos**: la ingesta de múltiples fuentes puede incrementarse con el tiempo.
- **Frecuencia de los datos**: Batch o streaming.
- **Formato de los datos**: estructurado (tablas), desestructurado (imágenes, audios, vídeos, etc.) o semiestructurado (JSON).

1.5.3 Herramientas de ingesta de datos

Necesitamos herramientas que faciliten la conexión a diferentes sistemas, como bases de datos SQL, APIs, servidores FTP. Entre las principales herramientas de ingesta de datos para ecosistemas Big Data podemos destacar:

- **Apache Sqoop** *https://sqoop.apache.org*: permite la transferencia bidireccional de datos entre Hadoop/Hive/HBase y bases de datos SQL.
- **Apache Flume** *https://flume.apache.org*: sistema de ingesta de datos semiestructurados o no estructurados sobre HDFS o HBase mediante una arquitectura basada en flujos de datos en streaming.
- **Apache Nifi** *https://nifi.apache.org*: herramienta que facilita una interfaz web que permite cargar datos de diferentes fuentes (tanto batch como streaming), los pasa por un flujo de procesos (mediante grafos dirigidos) para su tratamiento y transformación, y los vuelca en otra fuente.
- **Elastic Logstash** *https://www.elastic.co/es/logstash*: pensada inicialmente para la ingesta de logs en Elasticsearch, admite entradas y salidas de diferentes tipos.

- **AWS Glue** *https://aws.amazon.com/es/glue*: servicio gestionado por AWS para realizar tareas ETL desde la consola de AWS. Facilita el descubrimiento de datos y esquemas y normalmente se utiliza como almacenamiento de servicios como Amazon Athena o AWS Data Pipeline.

Por otro lado, existen sistemas de mensajería con funciones propias de ingesta, tales como:

- **Apache Kafka** *https://kafka.apache.org***:** sistema de intermediación de mensajes basado en el modelo publicador/suscriptor.
- **RabbitMQ** *https://www.rabbitmq.com*: sistema de colas de mensajes (Message Queue) que actúa de middleware entre productores y consumidores.
- **Amazon Kinesis** *https://aws.amazon.com/es/kinesis***:** homólogo de Kafka para la infraestructura Amazon Web Services.
- **Microsoft Azure Event Hubs** *https://azure.microsoft.com/es-es/products/event-hub*: servicio equivalente de Kafka para la infraestructura Microsoft Azure.
- **Google Pub/Sub** *https://cloud.google.com/pubsub*: servicio equivalente de Kafka para la infraestructura Google Cloud.

1.5.4 AWS Glue

AWS Glue también puede proponer transformaciones en función de los esquemas y los formatos de los datos identificados. De esta forma, podríamos transformar datos semiestructurados en datos estructurados y relacionales. Se puede usar también como un catálogo de datos con la ayuda del crawler y del servicio **Amazon Athena** *https://aws.amazon.com/athena*.

Además de las **ventajas** propias de los servicios Cloud, también encontramos las siguientes:

- El servicio AWS Glue se usa como herramienta serverless de integración de datos en la nube de AWS.
- Permite a los ingenieros de datos mover, combinar y transformar datos implementando pipelines ETL para realizar analítica o procesos de cálculo de manera sencilla.
- Proporciona multitud de conectores. Tanto para servicios propios de AWS como S3 o Redshift como con sistemas externos como Apache Kafka o MongoDB mediante diferentes conectores.
- Usa un modelo de pago por uso, en el que solamente se pagan los recursos en uso durante la ejecución de los trabajos.
- Para la monitorización de los trabajos, se puede integrar fácilmente con el servicio de AWS CloudWatch.

- El motor de ETL está basado en Apache Spark como motor de procesamiento distribuido para Big Data. Para ello, Glue permite implementar programas en los lenguajes de programación Python y Scala.

1.6 PROCESAMIENTO Y PREPARACIÓN

Una vez que los datos han sido recolectados y almacenados en algún repositorio central, el siguiente paso consiste en procesarlos y prepararlos para darles el formato adecuado para permitir el análisis y la ciencia de datos de una forma eficiente.

La etapa de procesamiento también suele llamarse "**Transformación**". Por un lado, está orientada a la limpieza de los datos ya que nos podemos encontrar con registros corruptos, con valores nulos o incluso repetidos. La limpieza busca eliminar o darles algún tratamiento a dichos datos con inconsistencias.

Por otro lado, en esta etapa se busca estandarizar los datos. Por ejemplo: al haber varios orígenes de datos, cada uno puede manipular las fechas con un formato específico e incluso con una zona horaria diferente. Entonces es necesario convertir esas fechas a algún formato estándar definido por la organización o por el proyecto donde estemos trabajando.

También, en el procesamiento y la transformación, se busca modelar los datos para darle una estructura entendible a los/as analistas y científicos/as de datos. El objetivo sería poder aplicar lógica para crear nuevas columnas o cruzar diferentes datos.

Por último, el equipo de ingeniería de datos debe entregar los datos en un formato óptimo para que el equipo de análisis y ciencia de datos pueda consumirlos y experimentar con ellos sin tanta latencia, por ejemplo. Al encontrarnos con grandes volúmenes de datos, el procesamiento suele enfocarse en herramientas que usen procesamiento a nivel de memoria para agilizar los tiempos de ejecución. En este punto podemos utilizar tecnologías como **Apache Spark, Apache Flink y Apache Beam.**

1.6.1 Tipos de procesamiento de datos

Procesamiento de datos puede ser un término amplio, aunque dentro de la ingeniería de datos el procesamiento implica la obtención de información a partir de los datos crudos. En primer lugar, hay que aplicar tareas "tradicionales" como tratar valores nulos, ya sea eliminándolos o reemplazándolos por otro valor, por ejemplo. Otra tarea tradicional puede ser la eliminación de registros duplicados. Las tareas en esta primera instancia implican obtener datos de calidad, aptos para continuar con tareas de proces

La siguiente etapa de procesamiento está vinculada a la organización o al negocio. Una vez que contamos con datos de calidad, el siguiente paso es enriquecer y obtener información valiosa para la organización. Aquí se pueden crear nuevas columnas que resulten de aplicar cálculos específicos, cruzar diferentes datos de diferentes fuentes, etc.

Para el procesamiento de datos, hay que tener en cuenta algunos conceptos entre los que podemos destacar:

- El procesamiento distribuido para el caso de grandes volúmenes de datos.
- El procesamiento batch y en streaming, cada uno ofrece tiempos de respuesta diferentes y operan sobre un lote de datos o sobre registros individuales, respectivamente.
- Data pipelines y orquestación, para asegurar que las tareas del proceso se ejecutan de forma ordenada y con robustez.

1.6.2 Procesamiento batch

El procesamiento de datos batch, o por lotes, opera sobre un volumen de datos definido, es decir sobre un lote de datos. Su ejecución se hace en intervalos programados, por ejemplo, cada hora, cada seis horas, cada día, etc. Además, la ejecución de estos procesos puede durar minutos u horas.

En este caso, los datos se recopilan y se almacenan hasta que se haya reunido una cantidad suficiente para procesarlos de una sola vez. Por ejemplo, si tenemos que trabajar con registros almacenados en un sistema, en lugar de procesar cada registro a medida que ocurre, el procesamiento batch recopila todos los registros en un lote, por ejemplo, cada hora o cada día, y luego los procesa juntos como un conjunto.

Este modo de procesamiento es útil cuando no se necesita respuesta en tiempo real y se toleran retrasos en el procesamiento. Es eficiente para trabajar con grandes volúmenes de datos, ya que se pueden aplicar optimizaciones y técnicas de procesamiento paralelo para acelerar el proceso.

1.6.3 Procesamiento streaming

El procesamiento de datos en streaming se refiere a la forma en que los datos se procesan de manera continua y en tiempo real a medida que se generan o se reciben. A diferencia del procesamiento por lotes, donde los datos se recopilan y se procesan en conjuntos, el procesamiento en streaming opera sobre los datos de forma inmediata, es decir, a medida que van fluyendo.

El procesamiento en streaming implica recibir, procesar y analizar continuamente los datos a medida que llegan. Esto se logra mediante el uso de sistemas y herramientas específicas, como Apache Kafka, Apache Flink o Apache Spark Streaming, que están diseñadas para manejar flujos continuos de datos.

A medida que los datos de streaming ingresan al sistema, se podrían aplicar operaciones en tiempo real para filtrar, transformar, enriquecer o agregar información a medida que los eventos se procesan. Estas operaciones pueden incluir cálculos, correlaciones, detección de anomalías o cualquier otro tipo de procesamiento requerido para extraer información valiosa de los datos en tiempo real.

Este modo de procesamiento es útil en situaciones donde se requiere baja latencia y una respuesta inmediata a los eventos que ocurren. Las principales aplicaciones que usan este tipo de procesamiento incluyen aquellas relacionadas con la monitorización de sistemas y aplicaciones en tiempo real.

1.7 ALMACENAMIENTO

Los datos deben concentrarse en algún sistema central. Para centralizar los datos crudos como archivos podemos usar tecnologías como Apache Hadoop, Amazon S3, Azure Data Lake Storage, Google Cloud Storage, Minio. Ahora bien, para permitir tareas de analítica, los datos deben entregarse en alguna base de datos OLAP como Apache Hive, Amazon Redshift, Google BigQuery, Azure Synapse, Apache Pinot, Apache Druid, Apache Impala, etc.

Al final de este capítulo analizaremos diferentes aspectos del almacenamiento de datos, en el contexto de Data Engineering donde trataremos sobre formatos de almacenamiento, como los basados en columnas y los basados en fila. Haremos foco en el primer tipo de formato, como por ejemplo Parquet, y su capacidad para comprimir el tamaño de los datos y permitir un análisis óptimo de los mismos.

A continuación, analizamos el término sobre Big Data y el almacenamiento distribuido. Big Data se define en término de 3 V: Volumen, Velocidad y Variedad. El almacenamiento distribuido es una técnica utilizada para almacenar y procesar grandes volúmenes de datos sobre clusters (servidores o computadores interconectadas entre sí), lo que permite una mayor escalabilidad y paralelismo en las operaciones.

1.7.1 Big Data

El término "Big Data" se refiere a conjuntos de datos que son tan grandes y complejos que no pueden gestionarse ni procesados fácilmente con herramientas tradicionales de procesamiento de datos. Estos conjuntos de datos grandes suelen caracterizarse por las llamadas "tres V":

- **Volumen**: se refiere a la gran cantidad de datos que se generan. Estos datos pueden alcanzar tamaños enormes, desde terabytes hasta petabytes o incluso exabytes.
- **Velocidad**: se refiere a la tasa a la cual se generan los datos. En algunos casos, los datos se generan en tiempo real, como las mediciones de sensores. El procesamiento de datos en tiempo real requiere tecnologías y enfoques especiales para garantizar que los datos se capturan y procesan en el menor tiempo posible.
- **Variedad**: se refiere a la diversidad de tipos y formatos de datos, desde estructurados y tabulares a semiestructurados y no estructurados, como textos, imágenes, videos, etc. Manejar esta variedad de datos requiere técnicas de procesamiento y análisis específicas.

- **Veracidad**: la veracidad se refiere a la calidad y confiabilidad de los datos, ya que pueden estar sujetos a problemas de calidad, como ruido, errores o inconsistencias. Es necesario aplicar técnicas de limpieza, normalización y validación de datos para garantizar su veracidad antes de su análisis.
- **Valor**: el valor se refiere al potencial de obtener información y conocimientos significativos a partir de los datos. El análisis de grandes volúmenes de datos puede revelar patrones, tendencias y correlaciones que pueden utilizarse para tomar decisiones informadas, descubrir oportunidades de negocio y mejorar la eficiencia operativa.

Dado el tamaño, la velocidad y la variedad de los datos, como ingenieros de datos debemos utilizar técnicas y herramientas especiales para gestionar y procesar estos datos. Esto incluye tecnologías como el almacenamiento y procesamiento distribuido, por medio de clusters como infraestructura, y el uso de frameworks y plataformas diseñadas específicamente para el Big Data, como Apache Hadoop y Apache Spark.

1.7.2 Almacenamiento distribuido

El almacenamiento distribuido es una técnica clave utilizada en un cluster, donde en lugar de almacenar todos los datos en una sola máquina, los datos se dividen en fragmentos más pequeños llamados particiones. Además, se crean copias o réplicas de cada partición y se distribuyen y almacenan en diferentes nodos del clúster. La distribución equilibrada de las particiones permite un acceso rápido y eficiente a los datos. La principal tecnología de Big Data que implementa el almacenamiento distribuido es Apache Hadoop y su sistema de archivos distribuido (HDFS: Hadoop Distributed File System).

En un contexto de Big Data, donde los volúmenes de datos son masivos y superan la capacidad de los sistemas tradicionales, el almacenamiento distribuido se vuelve fundamental. La capacidad de distribuir y procesar datos en clústers de servidores interconectados nos permite aprovechar al máximo los recursos y obtener resultados escalables y en menor tiempo. El almacenamiento distribuido nos brinda la flexibilidad necesaria para abordar los desafíos del Big Data y realizar análisis complejos en grandes conjuntos de datos.

1.7.3 Elegir una herramienta ETL para trabajar en Big Data

Al elegir una herramienta ETL (Extract, Transform, Load) para trabajar en entornos de Big Data, es importante considerar varios factores, como la escalabilidad, la integración con diferentes fuentes de datos, el rendimiento, y la facilidad de uso. A continuación, se menciona la lista de parámetros a considerar al elegir una herramienta ETL para Big Data.

- **Volumen de datos a gestionar.** ¿La herramienta está diseñada para la recuperación de datos desde una única fuente o desde múltiples fuentes? Las herramientas

utilizadas para la recuperación de datos de una sola fuente difieren de las diseñadas para la recuperación de datos de múltiples fuentes. En este punto sería recomendable medir el volumen y la velocidad de los datos a lo largo del tiempo.

- **Naturaleza de los datos.** Los datos pueden ser estructurados y no estructurados y provenir de diversas fuentes. En ocasiones, los datos deben procesarse en un formato que sea uniforme y comprensible para las herramientas analíticas. También se debe verificar si la herramienta ETL puede transformar ciertos datos producidos por otras herramientas en la organización. En este punto sería recomendable evaluar si es necesario extraer datos de fuentes no estructuradas como por ejemplo páginas web, email, etc.
- **Tareas que se espera que realice la herramienta.** Se debe comprender el tipo de datos que se espera que la herramienta recupere y procese, así como el punto final de entrega para todo el proceso ETL.

1.8 PROCESOS ETL (EXTRACT, TRANSFORM, LOAD)

ETL es el acrónimo de Extract, Transform, Load (extraer, transformar, cargar). Es un proceso fundamental en la ingeniería de datos que se utiliza para extraer datos de diversas fuentes, transformarlos de acuerdo con las necesidades específicas y cargarlos en un destino de almacenamiento, como un Data Warehouse.

Una vez realizada la extracción de los datos, se realizan una serie de transformaciones para limpiar, filtrar, combinar y estructurar los datos de acuerdo con el modelo de datos predefinido del Data Warehouse. Las transformaciones pueden incluir la normalización de datos, el cálculo de nuevas variables, la agregación de datos y la resolución de inconsistencias. El objetivo es garantizar la calidad y coherencia de los datos antes de cargarlos en el Data Warehouse.

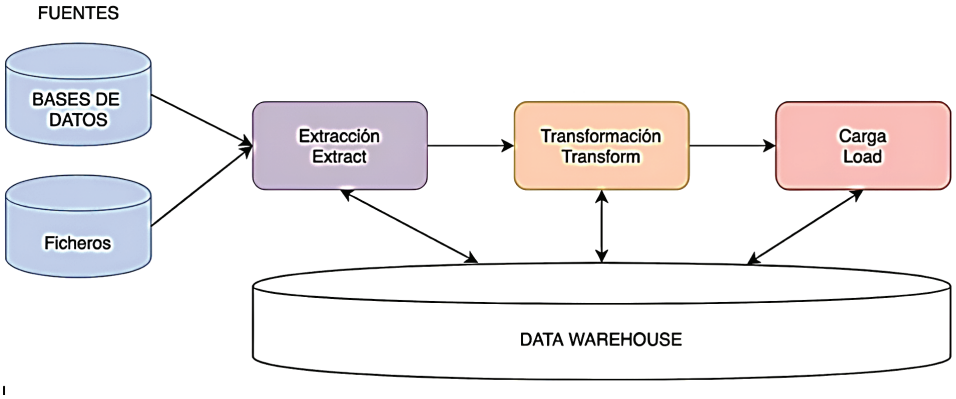

Figura 1.2. Esquema con las tres fases ETL: Extract, Transform, Load

Para entender la complejidad que puede haber en un proceso de este tipo, debemos entender primero el concepto de flujo o pipeline de datos. El flujo de datos es el conjunto de acciones que se realizan sobre un dato para moverlo, limpiarlo y procesarlo, hasta el punto en el que pueda consumirse. Los pipelines ETL se refieren a los procesos con las fases de extracción de datos de una fuente, su posterior transformación o filtrado y su carga en un sistema destino, como puede ser una base de datos o un data warehouse para su uso en procesos analíticos.

En ocasiones hablamos de las ETL de datos, donde movemos y procesamos datos de un lugar a otro para ser almacenados y tratados, los pipelines de datos incluyen procesos de ETL y, además, están orientados, no solo a la transferencia del dato, sino a aportar al final del pipeline un valor a negocio.

Dentro de las empresas se pueden manejar multitud de flujos de datos, desde aquellos que pueden generar sistemas de reporting, cuadros de mando o generar datos para actualización de modelos de Machine Learning, pero también flujos sencillos de captación y representación de datos para una aplicación. En cualquier caso, en muchas de estas ocasiones esos flujos de datos tienen tal periodicidad, criticidad y dependencias que, ante un fallo, restaurar el flujo puede ser un problema difícil de resolver.

Los procesos ETL permiten a las organizaciones recopilar en un único lugar todos los datos de los que pueden disponer. Ya comentamos que estos datos provienen de diversas fuentes, por lo que es necesario acceder a ellos y formatearlos para poder integrarlos. Además, es muy recomendable asegurar la calidad de los datos y su veracidad, para así evitar la creación de errores en los datos.

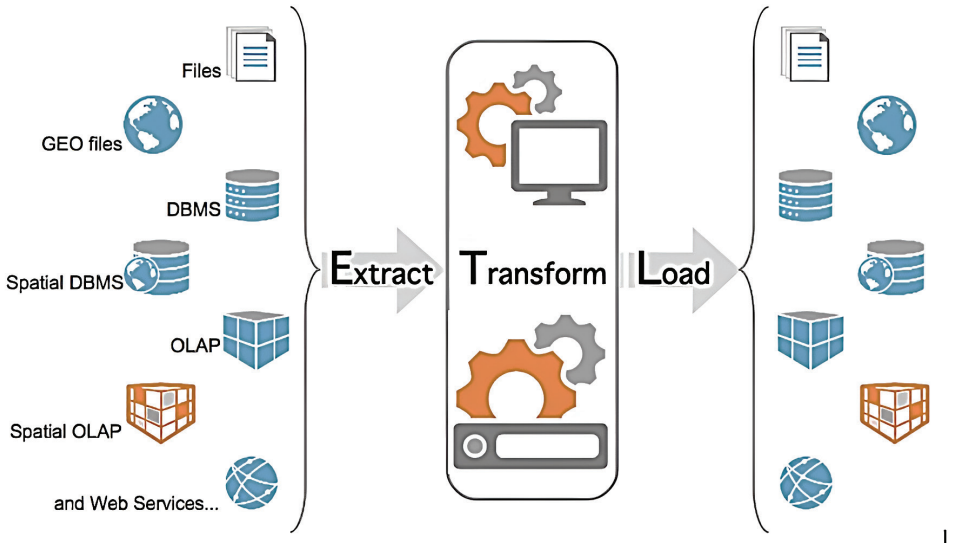

Figura 1.3. Fuente de datos para un proceso de ETL

Dada la variedad de posibilidades de representar la realidad en un dato, junto con la gran cantidad de datos almacenados en las diferentes fuentes de origen, los procesos ETL consumen muchos recursos asignados a un proyecto.

En los últimos años se han popularizado las herramientas ETL para gestionar los datos en tiempo real o streaming frente a los procesos de tipo Batch. Los casos de uso que requieren bajas latencias y obtener información de valor lo más rápido posible han aumentado mucho con los dispositivos IoT para Big Data y el enfoque hacia mejorar la experiencia de usuario.

1.8.1 Extracción

La fase de extracción consiste en la recuperación de información de varios sistemas de origen, como pueden ser RDBMS en forma de tabla o bien en formato JSON o XML, etc. Las fuentes de datos pueden ser muy variadas, como bases de datos relacionales o no relacionales, ficheros, aplicaciones SaaS, CRMs, ERPs, APIs, páginas web o sistemas de logs.

La primera característica deseable de un proceso de extracción es que debe ser un proceso rápido, ligero, causar el menor impacto posible, ser transparente para los sistemas operacionales e independiente de las infraestructuras. La segunda característica es que debe reducir al mínimo el impacto que se genera en el sistema origen de la información. Así pues, la extracción convierte los datos a un formato preparado para iniciar el proceso de transformación.

1.8.2 Transformación

Esta fase involucra varios procesos mediante los cuales los datos extraídos en la fase anterior se transforman en un formato útil y fácil de entender. En esta fase se espera realizar los cambios necesarios en los datos de manera que estos tengan el formato y contenido esperado. En concreto, la transformación puede comprender las siguientes tareas:

- Eliminar las entradas duplicadas, incompletas o incorrectas.
- Eliminar los campos innecesarios de los registros.
- Realizar un filtrado y una validación de los datos.
- Reordenar los datos no estructurados en datos estructurados.
- Unir los datos de diferentes fuentes con operaciones JOIN.
- Cruzar diferentes fuentes de datos para obtener una fuente diferente.
- Agregar información en función de alguna variable.
- Tomar parte de los datos para cargarlos.
- Transformar información para generar códigos, claves e identificadores.

1.8.3 Carga

Una vez transformados, los datos ya estarán listos para su carga. Esta es la fase final del proceso ETL, en la cual los datos se cargan en un almacén de datos. Existen dos tipos de cargas:

- **Completa:** todos los datos se mueven al almacén al mismo tiempo.
- **Incremental**: el movimiento de datos se produce en lotes, con una velocidad de refresco.

Como vemos en la imagen del Workflow de una ETL, generalmente se comienza extrayendo datos de bases de datos relacionales o ficheros, en la imagen se incluyen bases de datos relacionales como SQL Server, DB2 u Oracle, así como ficheros planos.

Esta fase es la encargada de almacenar los datos en el destino, un data warehouse o en cualquier tipo de base de datos. Por tanto, la fase de carga interactúa directamente con el sistema destino, y debe adaptarse al mismo para cargar los datos de manera satisfactoria.

1.8.4 Herramientas ETL

Las herramientas ETL (Extract, Transform, Load) son esenciales en la ingeniería de datos y se utilizan para extraer datos de diversas fuentes, transformarlos en un formato adecuado y cargarlos en un almacén de datos o sistema de destino. A continuación, analizamos algunas características de las herramientas ETL, junto con ejemplos de herramientas que podemos encontrar en el mercado:

- **Extracción de datos**: las herramientas ETL deben ser capaces de extraer datos de una amplia variedad de fuentes, incluidas bases de datos, sistemas en la nube, archivos planos, APIs web y más. **Apache Nifi** es una herramienta de código abierto que permite la extracción de datos de diversas fuentes y su enrutamiento a destinos específicos.
- **Transformación de datos**: las herramientas ETL permiten la limpieza, agregación, enriquecimiento y transformación de datos para que se ajusten a los requisitos de análisis y almacenamiento. **Apache Spark** es una plataforma de procesamiento de datos que se utiliza comúnmente para la transformación de datos en tiempo real y por lotes.
- **Manejo de datos en lote y en tiempo real**: algunas herramientas ETL pueden manejar tanto el procesamiento por lotes como el procesamiento en tiempo real para adaptarse a las necesidades de diferentes aplicaciones. **Apache Kafka Streams** permite la ingesta y transformación de datos en tiempo real.
- **Programación visual y código**: muchas herramientas ETL ofrecen interfaces de programación visual que permiten a los usuarios crear flujos de trabajo sin necesidad de escribir código, mientras que otras permiten la personalización mediante la escritura de código. **Apache NiFi** ofrece una interfaz de arrastrar y

soltar para crear flujos de trabajo ETL, mientras que **Apache Beam** permite la definición de flujos de trabajo ETL mediante código.

- **Planificación y orquestación**: las herramientas ETL suelen incluir capacidades de planificación y orquestación para ejecutar flujos de trabajo en horarios específicos o en respuesta a eventos. **Apache Airflow** es una plataforma de orquestación que se utiliza comúnmente para programar y administrar flujos de trabajo ETL.
- **Gestión de Errores y Tolerancia a Fallos**: las herramientas ETL deben ser capaces de manejar errores de manera efectiva y proporcionar mecanismos de tolerancia a fallos para garantizar la integridad de los datos. **Talend** es una herramienta ETL que ofrece características avanzadas de manejo de errores y recuperación.
- **Conexión a Almacenes de Datos**: las herramientas ETL deben ser compatibles con una variedad de almacenes de datos, como bases de datos relacionales, almacenes de datos en la nube, almacenes de columnas y más. **AWS Glue** es un servicio de ETL en la nube de Amazon que se integra con Amazon Redshift y Amazon S3, entre otros servicios.
- **Monitorización y Registro**: las herramientas ETL deben proporcionar capacidades de supervisión y registro para rastrear el rendimiento de los flujos de trabajo y detectar problemas. **Google Cloud Dataflow** ofrece una amplia gama de herramientas de supervisión y registro para flujos de trabajo ETL.

Las herramientas ETL son fundamentales en el proceso de gestión y transformación de datos, permitiendo a las organizaciones aprovechar sus datos de manera efectiva y eficiente. La elección de una herramienta ETL específica dependerá de las necesidades de tu proyecto, los sistemas de origen y destino, y las capacidades de procesamiento requeridas.

1.8.5 Herramientas de orquestación de flujos de datos

Las herramientas de orquestación de flujos de datos son software o plataformas diseñadas para gestionar y coordinar el procesamiento y movimiento de datos en un entorno de Big Data o cualquier contexto donde se requiera automatizar tareas relacionadas con el procesamiento de datos. Estas herramientas son fundamentales para garantizar la eficiencia, la confiabilidad y la escalabilidad de los flujos de datos en aplicaciones empresariales y científicas. Entre las principales herramientas para la orquestación de flujos de datos podemos destacar:

- **Apache NiFi**: NiFi es una plataforma de código abierto que proporciona una interfaz gráfica para diseñar flujos de datos y orquestar la recopilación, transformación y movimiento de datos entre sistemas diversos. Es útil para ingerir datos en tiempo real. Ofrece una interfaz web para definir flujos y un conjunto de conectores ya predefinidos para interactuar con fuentes y destinos de datos.

- **Apache Airflow:** Airflow es una plataforma de orquestación de código abierto diseñada principalmente para gestionar flujos de trabajo de procesamiento de datos, automatizar tareas y programar sus ejecuciones.
- **Apache Kafka**: Kafka es una plataforma de streaming de eventos que se utiliza comúnmente para la ingesta y el transporte de datos en tiempo real. Aunque no es una herramienta de orquestación en sí misma, es una parte fundamental de muchas arquitecturas de procesamiento de datos en tiempo real y a menudo se combina con otras herramientas de orquestación.
- **Apache Beam:** Beam es un modelo de programación unificado que permite la definición de flujos de datos portátiles y escalables. Puede ejecutarse en múltiples motores de procesamiento de datos, como Apache Flink, Apache Spark y Google Cloud Dataflow.
- **Apache Flink**: Flink es un motor de procesamiento de datos en tiempo real y por lotes que se puede utilizar para procesar y analizar flujos de datos continuos. Aunque no es una herramienta de orquestación, se usa junto con otras para implementar flujos de datos completos.
- **Apache Oozie**: centrada en su uso dentro del ecosistema Hadoop definiendo los flujos mediante XML. Oozie se centra en la creación de flujos de trabajo complejos, permitiéndonos tener trabajos activos por tiempo, por eventos o tener disponibilidad de datos según las situaciones en que la disponibilidad de los mismos pueda ser impredecible.
- **Google Cloud Dataflow**: Dataflow es un servicio completamente administrado en la nube de Google que permite crear flujos de datos paralelos y escalables utilizando el modelo Apache Beam. Es adecuado para implementaciones en la nube de flujos de datos.
- **AWS Step Functions**: este servicio de Amazon Web Services (AWS) permite crear y orquestar flujos de trabajo de datos sin servidor utilizando una interfaz visual. Puede utilizarse para coordinar diversas tareas y servicios de AWS.
- **Microsoft Azure Data Factory**: Azure Data Factory es un servicio de Microsoft Azure que permite crear, programar y orquestar flujos de datos en la nube. Se puede utilizar para mover datos entre servicios de Azure y sistemas locales, así como para realizar transformaciones de datos.
- **Talend**: Talend es una plataforma de integración de datos que proporciona capacidades de orquestación de flujos de datos, transformación de datos y migración de datos.
- **Kubeflow Pipelines:** si estás trabajando en un entorno de Kubernetes, Kubeflow Pipelines te permite orquestar flujos de trabajo de machine learning y procesamiento de datos en contenedores.
- Existen otras herramientas como pueden ser **Argo** *https://argoproj.github.io*, **Luigi** *https://luigi.readthedocs.io*, **Prefect** *https://www.prefect.io* o **Dagster** *https://dagster.io*.

Estas **plataformas de orquestación** de flujos de datos ofrecen capacidades muy útiles para programar, monitorear y ejecutar tareas en un pipeline de datos. Facilitan la definición y el control de tareas complejas, permiten la integración con diferentes sistemas y tecnologías, y proporcionan capacidades de monitoreo y administración para garantizar la confiabilidad y el rendimiento del pipeline. Los data pipelines son una parte fundamental en el flujo de trabajo de la ingeniería de datos y son cruciales en la obtención de información valiosa a partir de los datos.

1.9 TÉCNICAS DE EXTRACCIÓN DE DATOS

La extracción de datos es el punto de partida para cualquier proyecto de ingeniería de datos, donde se recopilan datos de diversas fuentes para transferirlos a un entorno de trabajo adecuado para su posterior procesamiento y análisis.

Además, analizaremos los formatos y las estructuras en las que suelen presentarse los datos. Y por último, detallaremos las técnicas de extracción más comunes utilizadas en Ingeniería de datos. Hablaremos de técnicas como la extracción incremental y la extracción "full" y explicaremos cuándo y por qué utilizar cada una. A grandes rasgos, una ingesta o recolección de datos se puede realizar de dos formas:

- Ingesta batch.
- En tiempo real.

1.9.1 Ingesta batch

La ingesta de datos de tipo batch es aquella que opera sobre un volumen de datos definido, también conocido como lote. Su ejecución se realiza durante un periodo de tiempo y de forma periódica, por ejemplo, cada 10 diez minutos, cada día o cada semana.

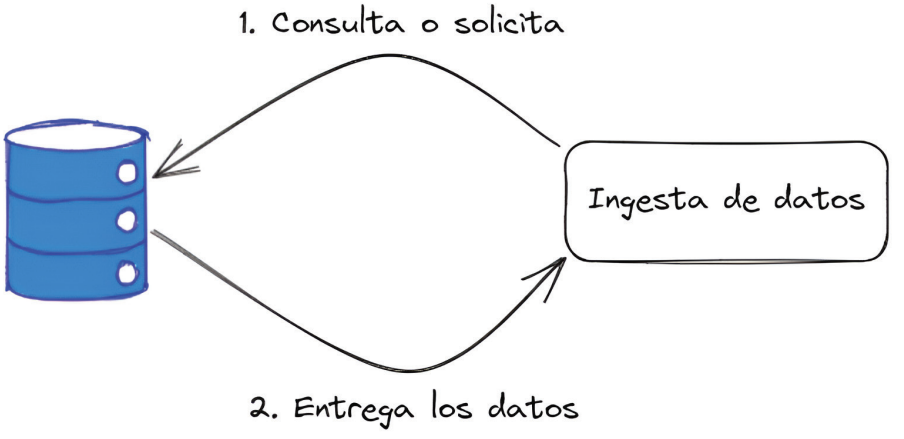

Figura 1.4. Ingesta de datos periódica sobre un conjunto de datos

A nivel de procesos batch podríamos diferenciar dos tipos de ingestas:

- **Ingesta completa.** Se caracteriza por extraer todos los datos de la fuente en cada ejecución del proceso de extracción y volcarlos de forma completa en el sistema de destino. El volcado de los datos puede sobreescribir lo que ya está disponible en el sistema de destino, o depositar los datos en otro archivo o directorio. Puede ser útil cuando se trabaja con fuentes de datos estáticas o cuando los requisitos del proyecto no permiten la identificación de cambios incrementales. Para este tipo de ingesta hay que tener en cuenta el volumen de datos, ya que en cada ejecución se toma una captura completa de la tabla o fuente de datos de origen.
- **Ingesta incremental (o delta).** Consiste en recolectar actualizaciones de la fuente de datos, ya sea por la inserción de nuevos registros o la modificación de los existentes. En vez de extraer todos los datos, se realizan consultas que seleccionan solo los datos nuevos o modificados desde la última extracción. El rastreo de cambios en esta técnica es posible si la fuente de datos cuenta con algún campo de identificación o con alguna marca de tiempo.

1.9.2 Ingesta en tiempo real

La ingesta de datos en tiempo real implica la captura de datos en tiempo real, a medida que se generan. En vez de esperar a la extracción programada, los datos se recolectan de inmediato, lo que permite obtener información actualizada en tiempo real. En la ingesta en tiempo real, los datos fluyen desde la fuente hacia el sistema destino y no existe una consulta o petición directamente por parte del recolector de datos.

Figura 1.5. Ingesta de datos en tiempo real sobre una cola de mensajes

Como ejemplo de datos en tiempo real, nos podemos encontrar con las mediciones de sensores o con publicaciones en una red social. Estos datos se denominan mensajes o eventos. En ingeniería de datos, los eventos o mensajes llegan a una "**cola de mensajes**" (message queue) o a una "plataforma de streaming de datos". Este tipo de tecnología se encarga de ordenar los datos recibidos y de asegurar que sean procesados una sola vez. Algunas tecnologías que permiten esto son Apache Kafka, Apache Pulsar, MQTT, RabbitMQ.

1.9.3 Change Data Capture

Existen ocasiones donde se necesita capturar cambios de una base de datos en tiempo real. CDC (Change Data Capture) es una técnica utilizada para identificar y capturar los cambios realizados en una base de datos en tiempo real.

El objetivo principal es detectar los cambios de manera eficiente y registrarlos para su procesamiento posterior. CDC permite capturar los cambios realizados en una fuente de datos sin tener que recorrer todos los datos nuevamente. En lugar de extraer todos los registros de una tabla, CDC registra solo los cambios realizados, como inserciones, actualizaciones o eliminaciones de registros individuales.

Existen diferentes formas de implementar CDC. Una podría ser un log de transacciones que encontramos en motores de bases de datos y registran las operaciones realizadas en la base de datos. Para implementar este tipo de CDC se utilizan tecnologías como **Debezium** *https://debezium.io* y **Apache Kafka** *https://kafka.apache.org*. Debezium está diseñado para leer los logs de transacciones de base de datos y convertir los cambios en eventos estructurados. Estos eventos se podrían enviar a **Apache Kafka,** para que los datos estén disponibles para su consumo casi al instante.

1.10 PIPELINES DE DATOS

Los desarrollos que podemos hacer como ingeniero/a de datos se podrían llamar "**Data Pipelines**". Se trata de un conjunto ordenado de procesos o rutinas que se encargan de obtener, procesar, verificar y entregar datos. El objetivo es automatizar los pipelines por medio de ciertas plataformas para que los procesos se ejecuten de forma periódica. A eso, se suma la orquestación, que consiste en lanzar la ejecución de los procesos del pipeline de forma ordenada por medio de alguna plataforma que, además tome alguna acción si se produce un error en algunas de las etapas del pipeline.

1.10.1 Definición de pipeline

Un pipeline es una construcción lógica que representa un proceso dividido en fases. Los pipelines de datos se caracterizan por definir el conjunto de pasos o fases y las tecnologías involucradas en un proceso de movimiento o procesamiento de datos. En su forma más simple, consisten en recoger los datos, almacenarlos y procesarlos, y construir algo útil con los datos.

Figura 1.6. Definición de pipeline de datos

Los pipelines de datos generalmente consisten en varias tareas o acciones que deben ejecutarse de forma ordenada para lograr el resultado deseado, mover datos de un lugar a otro. La salida de cada tarea o acción suele ser la entrada de la tarea siguiente.

Por lo general, las tareas deben ejecutarse en un orden específico, pudiendo haber casos en los que cada paso se ejecuta y finaliza antes de que comience el siguiente, asegurando un flujo secuencial y ordenado. También se pueden utilizar enfoques de procesamiento paralelo o distribuidos en un data pipeline, donde los pasos se ejecutan de manera concurrente. En este caso, los pasos pueden comenzar a ejecutarse en cuanto haya suficiente cantidad de datos disponibles para su procesamiento, sin esperar a que el anterior haya finalizado completamente.

Los pipelines de datos son necesarios ya que no debemos analizar los datos en los mismos sistemas donde se crean (principalmente para evitar problemas de rendimiento). Normalmente, los procesos de analítica son costosos computacionalmente, por lo que se separan para evitar perjudicar el rendimiento del servicio.

De esta forma, tenemos sistemas OLTP (sistemas de procesamiento transaccional online, como un CRM), encargados de capturar y crear datos, y de forma separada, sistemas OLAP (sistemas de procesamiento analítico, como un Data Warehouse), encargados de analizar los datos.

1.10.2 Fases de un pipeline de datos

Un pipeline es una construcción lógica que representa un proceso dividido en fases. Los pipelines de datos se caracterizan por definir el conjunto de pasos o fases y las tecnologías involucradas en un proceso de movimiento o procesamiento de datos. En su forma más simple, consisten en recoger los datos, almacenarlos y procesarlos, y construir algo útil con los datos. Los movimientos de datos entre estos sistemas involucran varias fases entre las que podemos destacar:

- **Ingesta**. Recogemos los datos y los enviamos a un topic de Apache Kafka.
- **Almacenamiento**. Kafka actúa aquí como un buffer para el siguiente paso.
- **Procesamiento**. Mediante una tecnología de procesamiento, que puede ser streaming o batch, leemos los datos del buffer.

- **Análisis**. Por ejemplo, mediante Spark realizamos la analítica sobre estos datos (haciendo cálculos, filtrados, agrupaciones de datos, etc..).
- **Visualización.** Finalmente, podemos visualizar los resultados obtenidos o almacenarlos en una base de datos NoSQL o un sistema de almacenamiento distribuido.

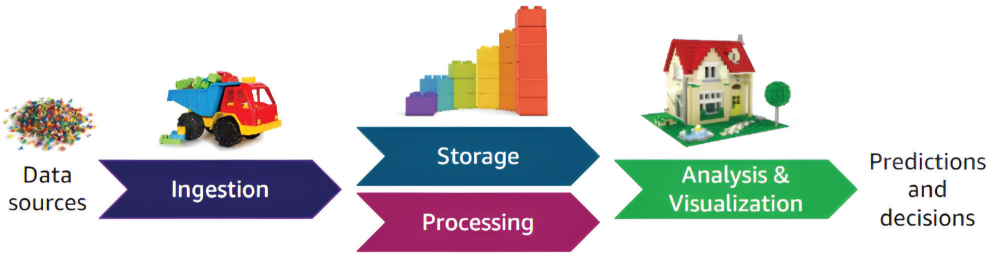

Figura 1.7. Fases de un pipeline de datos

Antes de realizar el análisis de los datos, va a ser muy normal tener que limpiar o normalizar los datos, ya sea porque las fuentes de datos, al ser distintas, utilicen diferentes codificaciones, o bien que haya datos sin rellenar o incorrectos. Estas transformaciones se conocen como **Data Wrangling** (manipulación de datos), término que engloba las acciones realizadas desde los datos en crudo hasta el estado final en el cual el dato cobra valor y sentido para los usuarios. El proceso de construcción de un data pipeline implica varias **fases** entre las que podemos destacar:

- **Extracción** de las fuentes de datos.
- **Transformación** de los datos, donde se aplican reglas y técnicas para filtrar, limpiar, enriquecer los datos.
- **Almacenamiento** de los datos, donde se cargan y persisten los datos ya procesados para que puedan ser consumidos por los interesados.
- **Orquestación**: un data pipeline también puede involucrar la orquestación de diferentes procesos y tareas en un flujo de trabajo secuencial. Esto implica programar y coordinar la ejecución de los diferentes pasos del pipeline, asegurándose de que se realicen en el orden correcto y que los datos fluyan de manera eficiente y confiable.

1.10.3 Pipeline iterativo

Este proceso de ingesta, almacenamiento, procesamiento y análisis es iterativo. Sobre una hipótesis que se nos plantee en negocio, comprobaremos los datos almacenados, y si no disponemos de la información necesaria, recogeremos nuevos datos. Estos nuevos datos pasarán por todo el pipeline, integrándose con los datos ya existentes. En la fase

de analítica, si no obtenemos el resultado esperado, nos tocará volver a la fase de ingesta para obtener o modificar los datos recogidos, y así, de forma iterativa, hasta producir el resultado esperado.

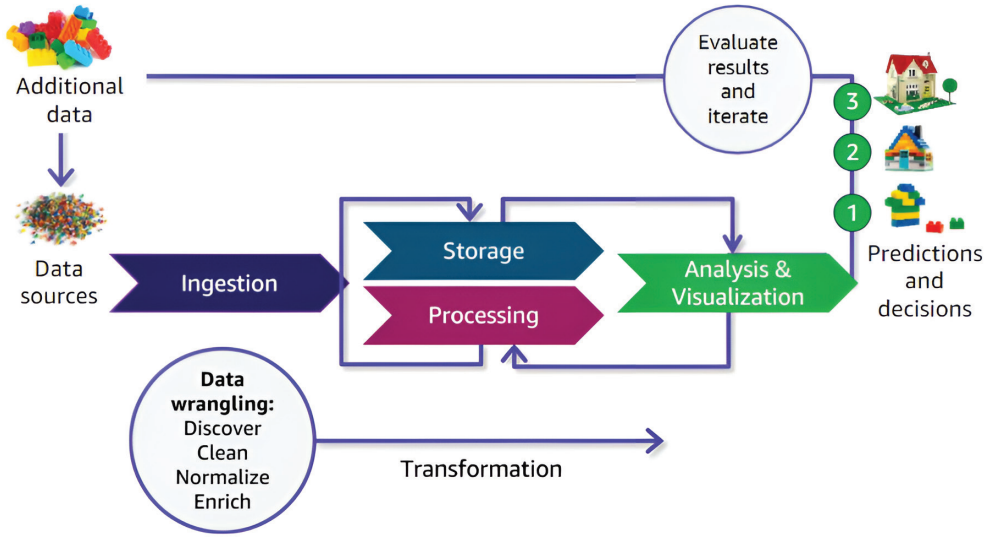

Figura 1.8. Fases de un pipeline iterativo

Es importante tener en cuenta varios aspectos clave para garantizar su eficiencia, confiabilidad y escalabilidad de los pipelines de datos. En cuanto a la gestión de errores y tolerancia a fallos, los data pipelines pueden enfrentar diferentes tipos de errores, como errores en la extracción de datos, errores de transformación o problemas de conectividad. Es fundamental implementar mecanismos de gestión de errores y tolerancia a fallos, como mecanismos de reintentos, registros o logs detallados de errores y alertas.

Estas medidas ayudan a garantizar la integridad y confiabilidad del pipeline, minimizando la pérdida de datos y el impacto en los procesos de análisis posteriores. Respecto al monitoreo y métricas, un data pipeline debe monitorearse con el objetivo de detectar posibles problemas o cuellos de botella. Es importante establecer métricas y alertas para supervisar el rendimiento, la latencia, el volumen de datos procesados y otros indicadores relevantes. Esto permite identificar y solucionar problemas de manera proactiva, asegurando que el pipeline funcione de manera óptima.

El último aspecto para destacar es la **modularidad** y **reutilización**. Es recomendable diseñar el data pipeline de manera modular y reutilizable. Esto implica dividir el pipeline en componentes más pequeños y funcionales, lo que facilita su mantenimiento, escalabilidad y reutilización en otros proyectos. El diseño modular también permite realizar pruebas y depuración más efectivas, ya que cada componente puede evaluarse de forma individual.

1.11 FUENTES DE DATOS

Una de las primeras tareas a realizar en un proyecto de Ingeniería de datos es la extracción de datos. Las organizaciones pueden contar con una variedad de fuentes, u orígenes, de datos y en general tenemos tres tipos principales de datos con los que un ingeniero de datos puede trabajar:

- **Estructurados**: los datos estructurados proceden principalmente de sistemas de origen basados en tablas, como una base de datos relacional o de un archivo plano, como un archivo separado por comas (CSV). El elemento principal de un archivo estructurado es que las filas y columnas se alinean de forma coherente en todo el archivo.
- **Semiestructurados**: los datos semiestructurados son datos como archivos de notación de objetos JavaScript (JSON), que pueden requerir acoplamiento antes de cargarlos en el sistema de origen. Cuando se aplanan, estos datos no tienen que ajustarse perfectamente a una estructura de tabla.
- **No estructurados**: los datos no estructurados incluyen datos almacenados como pares clave-valor que no cumplen los modelos relacionales estándar, y otros tipos de datos no estructurados que se usan normalmente incluyen formato de datos portátiles (PDF), documentos de procesador de texto e imágenes.

Como ingeniero de datos, algunas de las tareas principales incluyen integración de datos, transformación de datos y consolidación de datos.

- **Integración de datos:** la integración de datos implica establecer vínculos entre los servicios operativos y analíticos y los orígenes de datos para permitir el acceso seguro y confiable a los datos en varios sistemas. Por ejemplo, un proceso empresarial podría depender de datos que se distribuyen entre varios sistemas, y se requiere un ingeniero de datos para establecer vínculos para que se puedan extraer los datos necesarios de todos estos sistemas.
- **Transformación de datos**: los datos operativos normalmente deben transformarse en una estructura y formato adecuados para el análisis, a menudo como parte de un proceso de extracción, transformación y carga (ETL); aunque cada vez más se usa una variación en la que se extraen, cargan y transforman (ELT) los datos para ingerirlos rápidamente en un lago de datos y, a continuación, aplicar técnicas de procesamiento de "macrodatos" para transformarlos. Independientemente del enfoque usado, los datos están preparados para admitir las necesidades analíticas descendentes.
- **Consolidación de los datos**: la consolidación de datos es el proceso de combinar datos extraídos de varios orígenes de datos en una estructura coherente, normalmente para admitir análisis e informes. Normalmente, los datos de los sistemas operativos se extraen, transforman y cargan en almacenes analíticos, como un lago de datos o un almacenamiento de datos.

A continuación, vamos a describir algunas de las fuentes de datos disponibles en una organización, con las que podremos trabajar como ingeniero de datos.

1.11.1 Bases de datos

Es muy probable que muchos procesos de una organización registren sus datos en una base de datos relacional. Este tipo de base de datos también recibe el nombre de "transaccional" y es caracterizada por ser un sistema OLTP.

Las bases de datos transaccionales están diseñadas para recibir muchas escrituras, ya que los procesos de una organización suelen crear y modificar datos con mucha frecuencia. Este tipo de base de datos debe ser capaz de soportar esta carga de trabajo de forma óptima.

Las bases de datos relacionales organizan los datos mediante tablas, compuestas por filas y columnas. Las tablas de una base de datos se conectan entre sí mediante relaciones a partir de columnas en común.

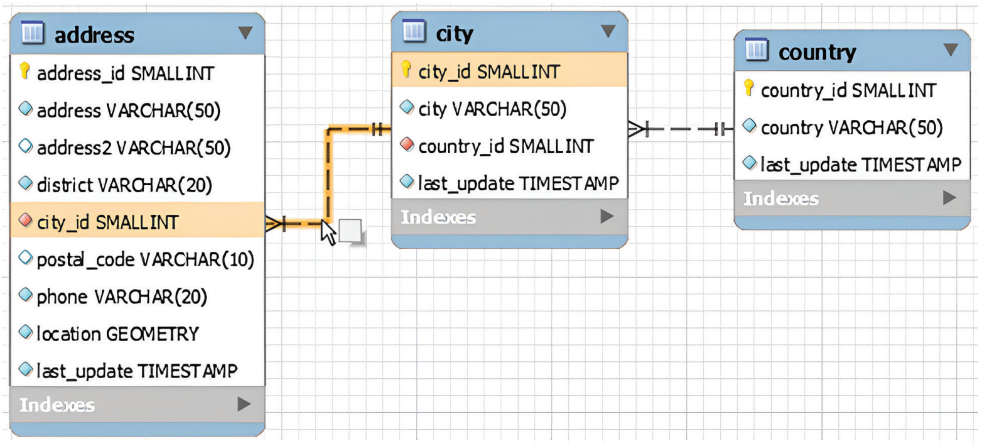

Figura 1.9. Esquema de una base de datos relacional

Las bases de datos se implementan con un software conocido como motor de base de datos. Existen diferentes motores de base de datos: MySQL, SQL Server, PostgreSQL. En el caso de bases de datos no relacionales, algunos motores son MongoDB, Redis, DynamoDB.

Para interactuar con la base de datos, por un lado, vamos a utilizar el lenguaje SQL para realizar consultas. Por otro lado, haremos uso de alguna herramienta ETL o de un lenguaje de programación para extraer los datos de las tablas. Cabe aclarar que estos procesos son posibles por medio de algún protocolo, como JDBC u ODBC, los cuales van a requerir la instalación de drivers específicos al motor de base de datos.

1.11.2 APIs

Como fuente de datos, una API actúa como un punto de acceso que permite obtener datos de una aplicación, servicio o sistema externo, sin tener que acceder directamente a la base de datos subyacente.

Además, las APIs suelen proporcionar diferentes "**endpoints**", que representan distintos conjuntos de datos o funcionalidades específicas, como operaciones de filtrado de datos. Las APIs normalmente hacen uso del protocolo HTTP, un protocolo de comunicación utilizado en la web para la transferencia de datos que está compuesto por una serie de métodos (GET, POST, PUT, DELETE, entre otros) que permiten realizar diferentes operaciones en los recursos a través de las URLs.

Por ejemplo, el método GET se utiliza para solicitar datos, el método POST se utiliza para enviar datos, el método PUT para actualizar y el método DELETE para eliminar. Las herramientas para extraer datos de una API pueden ser:

- Una plataforma ETL o de Integración, como Apache Nifi, que ofrece componentes predefinidos para conectarse y extraer datos de APIs.
- Algún lenguaje de programación, como Python, junto con alguna librería para realizar solicitudes HTTP a la API.
- Otras aplicaciones como Postman (basada en una interfaz gráfica) y cURL (basada en consola), que son útiles para pruebas sencillas.

1.12 FORMATOS ANALÍTICOS

Además de existir diversas fuentes que proveen datos, existen diferentes estructuras y formatos en las que se presentan. Una vez hemos recolectado y extraído datos de diferentes fuentes, los almacenamos en algún sistema centralizado para poder procesarlos, analizarlos y obtener valor de estos. Los datos se almacenan, en primera instancia, de forma cruda. A medida que los procesamos, limpiamos, transformamos, etc. iremos haciendo usos de formatos que optimicen el espacio y permitan un análisis eficiente.

Es muy probable que vayamos a manejar grandes volúmenes de datos y que nos encontremos con datos de millones de filas o registros, lo que se traduce en muchos gigabytes o terabytes de almacenamiento. En primer lugar, es posible reducir los tamaños de los datos trabajando con formatos binarios que comprimen los mismos. Seguramente, muchos de nosotros estamos acostumbrados a trabajar con formatos como .ZIP o .RAR. En el campo de la ingeniería, existen formatos similares que ofrecen varias ventajas. Los formatos más populares son **Parquet, Avro, ORC** y tienen en común lo siguiente:

- **Compresión**: reducen significativamente el tamaño de los archivos de datos.
- **Rendimiento**: ofrecen un buen rendimiento tanto en la lectura como en la escritura, frente a formatos de texto plano como CSV, JSON.

- **Autodescriptivos**: el esquema se almacena dentro del archivo de datos, de modo que las aplicaciones pueden entender los datos sin tener que depender de metadatos externos.
- **Evolución del esquema**: el esquema puede modificarse sin romper la compatibilidad con los archivos de datos existentes.

Ahora bien, además de estas similitudes, presentan diferencias en cuanto al formato de almacenamiento. **Avro** es un formato basado en filas (también denominado como row-based), mientras que **Parquet** y **ORC** son formatos columnares, o basados en columnas (column-based).

Los **formatos columnares** suelen ser más eficientes para cargas de trabajo analíticas, las cuales se caracterizan por leer o consultar grandes cantidades de datos. Mientras que los formatos basados en filas son más eficientes para cargas de trabajo operativas o transaccionales, las cuales consisten en muchas operaciones de escritura de datos.

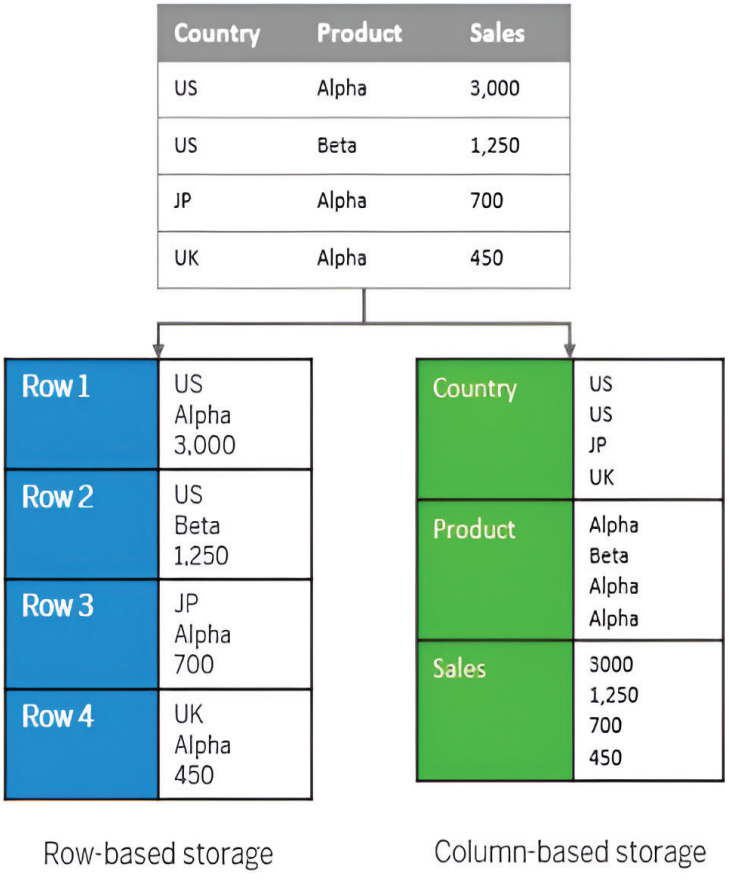

Figura 1.10. Esquema de una base de datos basada en filas vs columnas

Por ejemplo, si quisiéramos calcular el total de ventas, a partir del campo **Ventas (Sales)**:

- En el formato basado en filas (**row-based)**, hay que iterar por cada fila, obtener el campo "Sales" e ir acumulando ese número.
- En un formato basado en columnas (**column-based)**, los valores de ventas ya están disponibles en el mismo espacio y no es necesario iterar, solo es cuestión de aplicar la operación deseada.

Los formatos con los que estamos más familiarizados, como son CSV o JSON, se basan en filas, donde cada registro se almacena en una fila o documento. Estos formatos son más lentos en ciertas consultas y su almacenamiento no es óptimo.

En un formato basado en columnas, cada fila almacena toda la información de una columna. Al basarse en columnas, ofrece mejor rendimiento para consultas de determinadas columnas y/o agregaciones, y el almacenamiento es óptimo (como todos los datos de una columna son del mismo tipo, la compresión es mayor).

1.12.1 Apache Avro

Apache Avro *https://avro.apache.org* es un formato de almacenamiento basado en filas para Hadoop, utilizado para la serialización de datos, ya que es más rápido y ocupa menos espacio que otros formatos como JSON, debido a que la serialización de los datos se realiza en un formato binario compacto.

Cada fichero Avro almacena el esquema en la cabecera del fichero y luego están los datos en formato binario. Los esquemas se componen de tipos primitivos (null, boolean, int, long, float, double, bytes, y string) y compuestos (record, enum, array, map, union, y fixed).

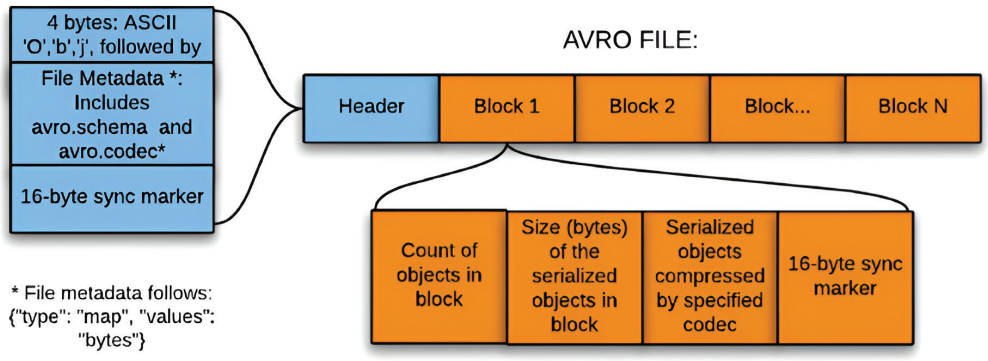

Figura 1.11. Esquema de un fichero en formato AVRO

Un ejemplo de esquema podría ser el siguiente fichero:

empleado.avsc

```
{
   “type” : “record”,
   “namespace” : “Empleados”,
   “name” : “Empleado”,
   “fields” : [
      { “name” : “Nombre” , “type” : “string” },
      { “name” : “Altura” , “type” : “float” }
      { “name” : “Edad” , “type” : “int” }
   ]
}
```

Para poder serializar y deserializar documentos **Avro** mediante Python, previamente debemos instalar la librería **avro-python3** *https://pypi.org/project/avro-python3*:

```
$ pip install avro-python3
Collecting avro-python3
  Downloading avro-python3-1.10.2.tar.gz (38 kB)
  Preparing metadata (setup.py) ... done
Building wheels for collected packages: avro-python3
  Building wheel for avro-python3 (setup.py) ... done
  Created wheel for avro-python3: filename=avro_python3-1.10.2-py3-none-any.whl
size=44009 sha256=df2f737cdda3fbbc33e52ed0603f420284d4a5e3f8ee6a3facd4c770a03d5
dee
  Stored in directory: /home/linux/.cache/pip/wheels/bb/73/e9/d273421f5723c4b-
f544dcf9eb097bda94421ef8d3252699f0a
Successfully built avro-python3
Installing collected packages: avro-python3
Successfully installed avro-python3-1.10.2
```

A continuación, vamos a realizar un ejemplo donde primero leemos un esquema de un archivo Avro, y con dicho esquema, escribiremos nuevos datos en un fichero. En el siguiente script abrimos el fichero escrito y leemos y mostramos los datos.

avro_schema.py

```
import avro
import copy
import json
from avro.datafile import DataFileReader, DataFileWriter
from avro.io import DatumReader, DatumWriter

# abrimos el fichero en modo binario y leemos el esquema
schema = avro.schema.parse(open(“empleado.avsc”, “rb”).read())

# escribimos un fichero a partir del esquema leído
```

```
with open('empleados.avro', 'wb') as f:
   writer = DataFileWriter(f, DatumWriter(), schema)
   writer.append({"nombre": "Carlos", "altura": 180, "edad": 44})
   writer.append({"nombre": "Juan", "altura": 175})
   writer.close()

# abrimos el archivo creado, lo leemos y mostramos línea a línea
with open("empleados.avro", "rb") as f:
   reader = DataFileReader(f, DatumReader())
   # copiamos los metadatos del fichero leído
   metadata = copy.deepcopy(reader.meta)
   # obtenemos el schema del fichero leído
   schemaFromFile = json.loads(metadata['avro.schema'])
   # recuperamos los empleados
   empleados = [empleado for empleado in reader]
   reader.close()

print(f'Schema de empleado.avsc:\n {schema}')
print(f'Schema del fichero empleados.avro:\n {schemaFromFile}')
print(f'Empleados:\n {empleados}')
```

Ejecución:

```
Schema de empleado.avsc:
 {"type": "record", "name": "empleado", "namespace": "Empleados", "fields":
[{"type": "string", "name": "nombre"}, {"type": "int", "name": "altura"},
{"type": ["null", "int"], "name": "edad", "default": null}]}
Schema del fichero empleados.avro:
 {'type': 'record', 'name': 'empleado', 'namespace': 'Empleados', 'fields':
[{'type': 'string', 'name': 'nombre'}, {'type': 'int', 'name': 'altura'},
{'type': ['null', 'int'], 'name': 'edad', 'default': None}]}
Empleados:
 [{'nombre': 'Carlos', 'altura': 180, 'edad': 44}, {'nombre': 'Juan', 'altura':
175, 'edad': None}]
```

1.12.2 Fastavro

Para trabajar con Avro y grandes volúmenes de datos, podríamos utilizar la librería **Fastavro** *https://github.com/fastavro/fastavro* la cual ofrece un mejor rendimiento ya que en vez de estar codificada en Python puro, tiene algunos fragmentos desarrollados utilizando **Cython**.

```
$ pip install fastavro
Collecting fastavro
  Obtaining dependency information for fastavro from https://files.pythonhosted.
org/packages/e9/00/c4e381c35eaee93cab9653bb345c5ffddc8ace5e84ddaa9cdb2f7a1022c0/
fastavro-1.9.4-cp310-cp310-manylinux_2_17_x86_64.manylinux2014_x86_64.whl.meta-
data
  Downloading fastavro-1.9.4-cp310-cp310-manylinux_2_17_x86_64.manylinux2014_
```

```
x86_64.whl.metadata (5.5 kB)
Downloading fastavro-1.9.4-cp310-cp310-manylinux_2_17_x86_64.manylinux2014_
x86_64.whl (3.1 MB)

   ---------------------------------------- 3.1/3.1 MB 3.2 MB/s eta 0:00:00
Installing collected packages: fastavro
Successfully installed fastavro-1.9.4
```

El siguiente ejemplo es similar al script anterior, con la diferencia que ahora utilizamos la librería **fastavro** para procesar el fichero y realizar operaciones de lectura y escritura sobre el mismo.

fastavro_schema.py

```
import fastavro
import copy
import json
from fastavro import reader

# abrimos el fichero en modo binario y leemos el esquema
with open("empleado.avsc", "rb") as f:
   schemaJSON = json.load(f)
schemaDict = fastavro.parse_schema(schemaJSON)

empleados = [{"nombre": "Carlos", "altura": 180, "edad": 44},
           {"nombre": "Juan", "altura": 175}]

# escribimos un fichero a partir del esquema leído
with open('empleadosf.avro', 'wb') as f:
   fastavro.writer(f, schemaDict, empleados)

# abrimos el archivo creado, lo leemos y mostramos línea a línea
with open("empleadosf.avro", "rb") as f:
   reader = fastavro.reader(f)
   # copiamos los metadatos del fichero leído
   metadata = copy.deepcopy(reader.metadata)
   # obtenemos el schema del fichero leído
   schemaReader = copy.deepcopy(reader.writer_schema)
   schemaFromFile = json.loads(metadata['avro.schema'])
   # recuperamos los empleados
   empleados = [empleado for empleado in reader]

print(f'Schema de empleado.avsc:\n {schemaDict}')
print(f'Schema del fichero empleadosf.avro:\n {schemaFromFile}')
print(f'Empleados:\n {empleados}')
```

Ejecución:

```
Schema de empleado.avsc:
 {'type': 'record', 'name': 'Empleados.empleado', 'fields': [{'name': 'nombre',
'type': 'string'}, {'name': 'altura', 'type': 'int'}, {'default': None, 'name':
'edad', 'type': ['null', 'int']}], '__fastavro_parsed': True, '__named_schemas':
{'Empleados.empleado': {'type': 'record', 'name': 'Empleados.empleado', 'fields':
[{'name': 'nombre', 'type': 'string'}, {'name': 'altura', 'type': 'int'}, {'de-
fault': None, 'name': 'edad', 'type': ['null', 'int']}]}}}
Schema del fichero empleados_fastavro.avro:
 {'type': 'record', 'name': 'Empleados.empleado', 'fields': [{'name': 'nombre',
'type': 'string'}, {'name': 'altura', 'type': 'int'}, {'default': None, 'name':
'edad', 'type': ['null', 'int']}]}
Empleados:
 [{'nombre': 'Carlos', 'altura': 180, 'edad': 44}, {'nombre': 'Juan', 'altura':
175, 'edad': None}]
```

En el siguiente ejemplo vamos a leer un fichero CSV mediante la librería **Pandas** *https://pandas.pydata.org*, y almacenaremos el resultado del procesamiento en un fichero en formato Avro con la librería **fastavro** *https://pypi.org/project/fastavro*.

pandas_fastavro.py

```
import pandas as pd
import fastavro
import copy
import json
from fastavro import writer, parse_schema

# Leemos el csv mediante pandas
df = pd.read_csv('ventas.csv',sep=';')

# Definimos el esquema
schema = {
   'name': 'Ventas',
   'namespace' : 'Ventas',
   'type': 'record',
   'fields': [
       {'name': 'ProductID', 'type': 'int'},
       {'name': 'Date', 'type': 'string'},
       {'name': 'Zip', 'type': 'string'},
       {'name': 'Units', 'type': 'int'},
       {'name': 'Revenue', 'type': 'float'},
       {'name': 'Country', 'type': 'string'}
   ]
}
schemaParseado = parse_schema(schema)

# Convertimos el Dataframe a una lista de diccionarios
records = df.to_dict('records')
```

```
# Persistimos en un fichero avro
with open('ventas.avro', 'wb') as f:
   writer(f, schemaParseado, records,'deflate')

# abrimos el archivo creado, lo leemos y mostramos línea a línea
with open("ventas.avro", "rb") as f:
   reader = fastavro.reader(f)
   # copiamos los metadatos del fichero leído
   metadata = copy.deepcopy(reader.metadata)
   # obtenemos el schema del fichero leído
   schemaReader = copy.deepcopy(reader.writer_schema)
   schemaFromFile = json.loads(metadata['avro.schema'])
   # Lee los primeros 10 registros
   primeros_10_ventas = [venta for _, venta in zip(range(10), reader)]

print(f'Schema del fichero ventas.avro:\n {schemaFromFile}')
print(f'Ventas:\n {primeros_10_ventas}')
```

Ejecución:

```
Schema del fichero ventas.avro:
 {'type': 'record', 'name': 'Ventas.Ventas', 'fields': [{'name': 'ProductID',
'type': 'int'}, {'name': 'Date', 'type': 'string'}, {'name': 'Zip', 'type':
'string'}, {'name': 'Units', 'type': 'int'}, {'name': 'Revenue', 'type':
'float'}, {'name': 'Country', 'type': 'string'}]}
Ventas:
 [{'ProductID': 725, 'Date': '1/15/1999', 'Zip': '41540         ', 'Units': 1,
'Revenue': 115.5, 'Country': 'Germany'}, {'ProductID': 787, 'Date': '6/6/2002',
'Zip': '41540         ', 'Units': 1, 'Revenue': 314.8999938964844, 'Country':
'Germany'}, {'ProductID': 788, 'Date': '6/6/2002', 'Zip': '41540         ',
'Units': 1, 'Revenue': 314.8999938964844, 'Country': 'Germany'}, {'Produc-
tID': 940, 'Date': '1/15/1999', 'Zip': '22587         ', 'Units': 1, 'Reve-
nue': 687.7000122070312, 'Country': 'Germany'}, {'ProductID': 396, 'Date':
'1/15/1999', 'Zip': '22587         ', 'Units': 1, 'Revenue': 857.0999755859375,
'Country': 'Germany'}, {'ProductID': 734, 'Date': '4/10/2003', 'Zip': '22587
', 'Units': 1, 'Revenue': 330.70001220703125, 'Country': 'Germany'}, {'Pro-
ductID': 769, 'Date': '2/15/1999', 'Zip': '22587         ', 'Units': 1, 'Re-
venue': 257.20001220703125, 'Country': 'Germany'}, {'ProductID': 499, 'Date':
'1/15/1999', 'Zip': '12555         ', 'Units': 1, 'Revenue': 846.2999877929688,
'Country': 'Germany'}, {'ProductID': 2254, 'Date': '1/15/1999', 'Zip': '40217
', 'Units': 1, 'Revenue': 57.70000076293945, 'Country': 'Germany'}, {'Produc-
tID': 31, 'Date': '5/31/2002', 'Zip': '40217         ', 'Units': 1, 'Revenue':
761.2000122070312, 'Country': 'Germany'}]
```

1.12.3 Comprimiendo los datos

Para trabajar con Avro y grandes volúmenes de datos, podríamos utilizar la librería **Fastavro**. Fastavro soporta dos tipos de compresión: gzip (mediante el algoritmo **deflate**) y **snappy**.

```
#Persistimos en un fichero avro
with open('ventas.avro', 'wb') as f:
      writer(f, schemaParseado, records,'deflate')
```

Snappy *https://pypi.org/project/python-snappy* es una biblioteca de compresión y descompresión de datos de gran rendimiento que se utiliza con frecuencia en proyectos Big Data, la cual hemos de instalar previamente con el siguiente comando:

```
$ pip install python-snappy

Collecting python-snappy
  Downloading python_snappy-0.6.1-cp38-cp38-manylinux_2_12_x86_64.manylinux2010_
x86_64.whl (56 kB)
     ———————————————————————————————————————————————————————————————
———————— 56.1/56.1 kB 1.1 MB/s eta 0:00:00
Installing collected packages: python-snappy
Successfully installed python-snappy-0.6.1
```

Para indicar el tipo de compresión, únicamente hemos de añadir un parámetro extra con el algoritmo de compresión en la función/constructor de persistencia:

```
# Persistimos en un fichero avro con el tipo de compresión 'snappy'
with open('ventas.avro', 'wb') as f:
      writer(f, schemaParseado, records,'snappy')
```

1.12.4 Parquet

Apache Parquet es un formato de almacenamiento basado en columnas, con soporte para muchos de los frameworks de procesamiento de datos, así como lenguajes de programación. De la misma forma que Avro, se trata de un formato de datos auto-descriptivo, de manera que embebe el esquema o estructura de los datos con los propios datos en sí.

Una de sus principales características es que permite acelerar y optimizar los tiempos de consulta sobre los datos, ya que sólo pueden leer las columnas necesarias para una consulta concreta. Por eso, está pensado para cargas de trabajo analítica donde, generalmente, se realizan cálculos sobre un conjunto de columnas en particular.

En la práctica, los lenguajes de programación y las librerías ya cuentan con métodos y funcionalidades predefinidas para manipular formatos como Parquet. Por ejemplo, la librería Pandas de Python cuenta con el método **to_parquet** para el almacenamiento de los datos *https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.to_parquet.html*. Esto es posible, ya que son formatos abiertos y permiten la interoperabilidad con diferentes tipos de herramientas.

El formato parquet ofrece un ratio de compresión muy alto (mediante Snappy ronda el 75%), además, solo se recorren las columnas necesarias en cada lectura, lo que reduce las operaciones de entrada/salida en disco.

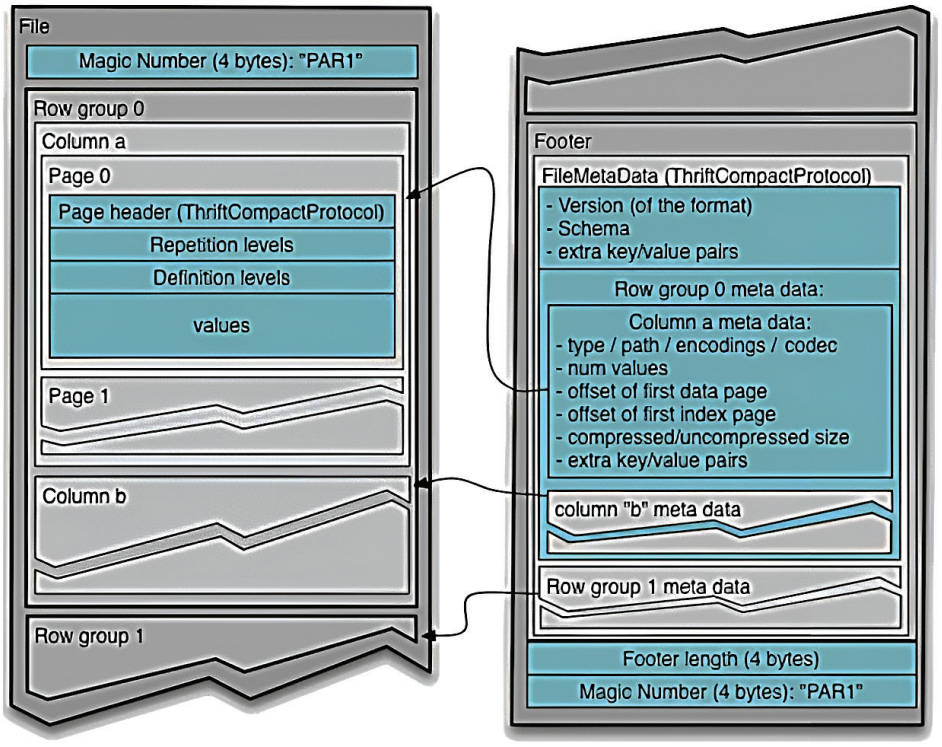

Figura 1.12. Esquema de un fichero en formato Parquet

Cada fichero Parquet almacena los datos en binario organizados en grupos de filas. Para cada grupo de filas (row group), los valores de los datos se organizan en columnas, lo que facilita la compresión a nivel de columna.

La columna de metadatos de un fichero Parquet se almacena al final del fichero, lo que permite que las escrituras sean rápidas con una única pasada. Los metadatos pueden incluir información como los tipos de datos, esquemas de codificación/compresión, estadísticas, nombre de los elementos, etc…

1.12.5 Interactuando con Parquet mediante Pyarrow

Para interactuar con el formato Parquet mediante Python, la librería más utilizada es la que ofrece **Apache Arrow** *https://arrow.apache.org*, en concreto la librería **PyArrow** *https://arrow.apache.org/docs/python*.

```
$ pip install pyarrow
```

Apache Arrow usa un tipo de estructura en formato tabla para almacenar los datos de forma bidimensional, muy similar al formato Dataframe de Pandas. La documentación de PyArrow dispone de un libro de recetas *https://arrow.apache.org/cookbook/py* con ejemplos con código para los diferentes casos de uso que se nos puedan plantear.

Vamos a simular el mismo ejemplo que hemos realizado previamente mediante Avro, y vamos a crear un fichero en formato JSON con empleados, y tras persistir en formato Parquet, lo vamos a recuperar:

empleados_parquet.py

```
import pyarrow.parquet as pq
import pyarrow as pa

# 1.- Definimos el esquema
schema = pa.schema([ ('nombre', pa.string()),
                     ('altura', pa.int32()),
                     ('edad', pa.int32())  ])

# 2.- Almacenamos los empleados por columnas
empleados = {"nombre": ["Carlos", "Juan"],
             "altura": [180, 44],
             "edad": [None, 34]}

# 3.- Creamos una tabla Arrow y la persistimos mediante Parquet
tabla = pa.Table.from_pydict(empleados, schema)
pq.write_table(tabla, 'empleados.parquet')

# 4.- Leemos el fichero generado
table2 = pq.read_table('empleados.parquet')
schemaFromFile = table2.schema

print(f'Schema del fichero empleados.parquet:\n{schemaFromFile}\n')
print(f'Tabla de Empleados:\n{table2}')
```

Ejecución:

```
Schema del fichero empleados.parquet:
nombre: string
altura: int32
edad: int32

Tabla de Empleados:
pyarrow.Table
nombre: string
altura: int32
edad: int32
----
nombre: [["Carlos","Juan"]]
altura: [[180,44]]
edad: [[null,34]]
```

En el caso del uso de Pandas el código se simplifica. El siguiente ejemplo estamos leyendo un archivo en formato Parquet y lo convertimos a un dataframe de Pandas:

parquet_pandas.py

```
import pyarrow.parquet as pq
empleados = pq.read_table('empleados.parquet')
empleados_df = empleados.to_pandas()
print(type(empleados_df))
print(empleados_df)
```

Ejecución:

```
<class 'pandas.core.frame.DataFrame'>
   nombre  altura  edad
0  Carlos    180   NaN
1  Juan       44   34.0
```

También podríamos convertir un csv a formato parquet a través del método **to_parquet** que ofrece la librería pandas.

- *https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.to_parquet.html.*

csv_parquet.py

```
import pandas as pd
import pyarrow.parquet as pq

df = pd.read_csv('ventas.csv',sep=';')

# A partir de un DataFrame, persistimos los datos
df.to_parquet('ventas.parquet')

# Leemos el fichero generado
table = pq.read_table('ventas.parquet')
schemaFromFile = table.schema

print(f'Schema del fichero ventas.parquet:\n{schemaFromFile}\n')
print(f'Tabla de ventas:\n{table}')
```

Ejecución:

```
Schema del fichero ventas.parquet:
ProductID: int64
Date: string
Zip: string
Units: int64
```

```
Revenue: double
Country: string
-- schema metadata --
pandas: '{"index_columns": [{"kind": "range", "name": null, "start": 0, "' + 929

Tabla de ventas:
pyarrow.Table
ProductID: int64
Date: string
Zip: string
Units: int64
Revenue: double
Country: string
----
ProductID: [[725,787,788,940,396,...,726,758,1076,1182,702],[714,758,739,895,757
,...,730,1105,1936,2263,2331],...,[2331,206,506,1995,2049,...,1077,1078,1129,221
5,1009],[734,978,978,359,733,...,2225,1114,1997,1060,1416]]
Date: [["1/15/1999","6/6/2002","6/6/2002","1/15/1999","1/15/1999",...,"11/29/200
2","11/29/2002","3/30/2001","3/30/2001","3/31/2001"],["3/31/2001","3/31/2001","3
/31/2001","3/31/2001","3/31/2001",...,"3/5/2001","3/5/2001","3/5/2001","3/5/2001
","3/5/2001"],...,["3/31/2011","3/31/2011","3/31/2011","3/31/2011","3/31/2011",.
..,"3/31/2010","3/31/2010","3/31/2010","3/31/2010","4/1/2010"],["4/1/2010","4/1/
2010","4/1/2010","4/1/2010","4/1/2010",...,"1/25/2015","1/25/2015","5/7/2015","5
/7/2015","5/7/2015"]]
Zip: [["41540         ","41540         ","41540        ","22587         ","22587
",...,"7010          ","7010          ","5080         ","5030          ","5050
"],["7049            ","7220          ","7187         ","7150          ","5030
",...,"75831 CEDEX 17 ","75725 CEDEX 15 ","75702 CEDEX 13 ","75929 CEDEX 19
","75710 CEDEX 15 "],...,["10243       ","10247       ","10409      ","12163
","10365       ",...,"75499 CEDEX 10 ","75554 CEDEX 11 ","75387 CEDEX 08
","75472 CEDEX 10 ","75484 CEDEX 10 "],["75504 CEDEX 15 ","75389 CEDEX 08
","75509 CEDEX 15 ","75394 CEDEX 08 ","75504 CEDEX 15 ",...,"75201 CEDEX 13
","75021 CEDEX 01 ","75012      ","75215 CEDEX 16 ","75055 CEDEX 01 "]]
Units: [[1,1,1,1,1,...,1,1,1,1,1],[1,1,1,1,1,...,1,1,1,1,1],...,[1,1,1,1,1,...,1
,1,1,1,2],[1,1,1,1,1,...,1,2,1,1,1]]
Revenue: [[115.5,314.9,314.9,687.7,857.1,...,110.2,73.5,254.6,173.2,288.7],[162.
7,84,173.2,577.5,84,...,309.7,246.7,280.8,299.2,567],...,[697.7,871.5,1244.2,446
.2,456.7,...,341.2,351.7,461.9,393.7,225.7],[398.9,782.2,782.2,1144.2,398.9,...,
63,404.1,787.5,157.4,472.4]]
Country: [["Germany","Germany","Germany","Germany","Germany",...,"Mexico
","Mexico ","Mexico ","Mexico ","Mexico "],["Mexico ","Mexico ","Mexico ","Mexi-
co ","Mexico ",...,"France ","France ","France ","France ","France "],...,["Ge
rmany","Germany","Germany","Germany","Germany",...,"France ","France ","Fran-
ce ","France ","France "],["France ","France ","France ","France ","France
",...,"France ","France ","France ","France ","France "]]
```

1.12.6 Apache ORC

Apache ORC *https://orc.apache.org* es un formato de datos basado en columnas, optimizado para la lectura, escritura y procesamiento de datos. Los ficheros ORC se componen de tiras de datos (stripes), donde cada tira contiene un índice, los datos de la

fila y un pie (con estadísticas como la cantidad, máximos y mínimos y la suma de cada columna convenientemente cacheadas).

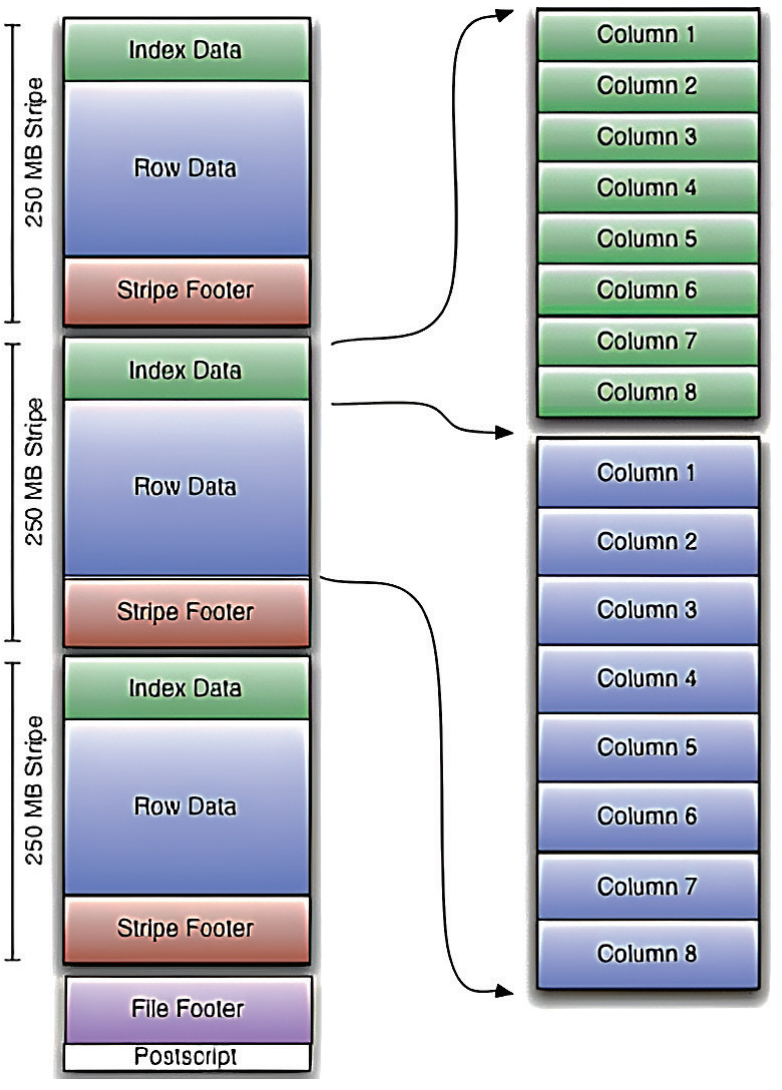

Figura 1.13. Esquema de un fichero en formato ORC

Para crear un archivo ORC y leerlo, volvemos a necesitar la librería **PyArrow**. Así pues, para la escritura de datos, por ejemplo, desde un Dataframe:

csv_orc.py

```
import pandas as pd
import pyarrow as pa
import pyarrow.orc as orc

df = pd.read_csv('pdi_sales.csv',sep=';')

table = pa.Table.from_pandas(df, preserve_index=False)
orc.write_table(table, 'pdi_sales.orc')

df_orc = pd.read_orc('pdi_sales.orc')
print(df_orc)
```

Ejecución:

```
$ python csv_orc.py
        ProductID         Date               Zip     Units  Revenue   Country
0         725       1/15/1999             41540         1    115.5   Germany
1         787        6/6/2002            41540          1    314.9   Germany
2         788        6/6/2002            41540          1    314.9   Germany
3         940       1/15/1999             22587         1    687.7   Germany
4         396       1/15/1999             22587         1    857.1   Germany
...       ...             ...                ...       ...      ...       ...
841142   2225       1/25/2015    75201 CEDEX 13         1     63.0    France
841143   1114       1/25/2015    75021 CEDEX 01         2    404.1    France
841144   1997        5/7/2015    75012                  1    787.5    France
841145   1060        5/7/2015    75215 CEDEX 16         1    157.4    France
841146   1416        5/7/2015    75055 CEDEX 01         1    472.4    France

[841147 rows x 6 columns]
```

1.12.7 Comparando formatos

Los formatos basados en filas ofrecen un rendimiento mayor en las escrituras que en las lecturas, ya que añadir nuevos registros es más sencillo. Si sólo vamos a hacer consultas sobre un subconjunto de las columnas, entonces un formato columnas se comportará mejor, ya que no necesita recuperar los registros enteros.

Respecto a la compresión, entendiendo que ofrece una ventaja a la hora de almacenar y transmitir la información, es útil cuando trabajamos con un gran volumen de datos. Los formatos basados en columnas ofrecen un mejor rendimiento ya que todos los datos del mismo tipo se almacenan de forma continua en memoria, lo que permite una mayor eficiencia en la compresión.

Respecto a la evolución del esquema, con operaciones como añadir o eliminar columnas o cambiar su nombre, la mejor decisión es decantarse por Avro. Además, al tener el esquema en JSON facilita su gestión y permite que tengan más de un esquema.

Si nuestros documentos tienen una estructura compleja compuesta por columnas anidadas y normalmente realizamos consultas sobre un subconjunto de las subcolumnas, la elección debería ser Parquet. Finalmente, hay que recordar que ORC está especialmente enfocado a su uso con Hive, mientras que Spark tiene un amplio soporte para Parquet y que, si trabajamos con Apache Kafka, Avro es una buena elección.