Data Engineering Design Patterns

Chapter 2. Data Ingestion Design Patterns

Bartosz Konieczny

November 22, 2025

Capítulo 2. Patrones de Diseño de Ingesta de Datos

Los sistemas de ingeniería de datos raramente generan sus propios datos; su primera etapa es casi siempre la adquisición de datos de diversos productores. Trabajar con estos productores (otros equipos, pipelines u organizaciones) es un desafío, ya que cada uno tiene sus propias restricciones técnicas y de negocio.

Es una tarea de adaptación obligatoria. Sin una ingesta de datos eficaz, las cargas de trabajo de análisis y ciencia de datos no se alimentan. Peor aún, se pueden obtener datos incompletos o corruptos, lo que genera quejas de los consumidores y requiere costosos procesos de restauración y *backfilling*.

Este capítulo se enfoca en los patrones de diseño de ingesta de datos, una tarea clave para el éxito de cualquier sistema de datos. Se abordan los siguientes temas:

- **Escenarios de carga comunes:** Cargas completas e incrementales.
- **Replicación de datos:** Copia de datos con y sin transformación, abordando problemas de privacidad.
- **Aspectos técnicos:** Se discute cuándo iniciar el proceso de ingesta (preparación de datos), cómo mejorar la experiencia del usuario solucionando el problema de los archivos pequeños (compactación de datos) y cómo manejar la incertidumbre en la llegada de los datos.

Carga Completa (*Full Load*)

El patrón de diseño de carga completa se refiere al escenario de ingesta que trabaja sobre un conjunto de datos completo cada vez. Es útil en situaciones como la inicialización de una base de datos o la generación de un conjunto de datos de referencia.

Patrón: Cargador Completo (*Full Loader*)

Es uno de los patrones más sencillos, pero su simple construcción de dos pasos tiene sus escollos.

Problema Se está configurando la capa Plata y una de las transformaciones requiere información de un proveedor de datos externo. Este conjunto de datos cambia pocas veces por semana, evoluciona lentamente y no supera el millón de filas. El problema es que el proveedor no ofrece ningún atributo (como una fecha de última actualización) para detectar qué filas han cambiado desde la última ingesta.

Solución La falta de un valor para detectar cambios hace que el patrón *Full Loader* sea la solución ideal. La implementación más simple se basa en dos pasos: **extraer y cargar (EL)**. Utiliza comandos nativos de los almacenes de datos para exportar de una base de datos e importar a otra. Este enfoque es ideal para almacenes de datos homogéneos, ya que no requiere transformación.

- **Trabajos de Passthrough:** Los trabajos de extracción y carga también se conocen como trabajos de *passthrough* porque los datos simplemente pasan a través del pipeline, desde la fuente hasta el destino.

Si los almacenes de datos son heterogéneos, el pipeline se convierte en un trabajo de **extracción, transformación y carga (ETL)**, donde se necesita una fina capa de transformación para adaptar los formatos.

Consecuencias

- **Volumen de datos:** Las implementaciones de *Full Loader* suelen ser trabajos por lotes (*batch*). Si el volumen de datos crece lentamente, la infraestructura funcionará sin problemas. Sin embargo, si el conjunto de datos evoluciona dinámicamente (por ejemplo, duplica su tamaño), los recursos de cómputo estáticos pueden ralentizar el proceso o incluso provocar fallos. Para mitigar esto, se pueden aprovechar las capacidades de autoescalado de la capa de procesamiento de datos.

- **Consistencia de los datos:** El segundo riesgo es la pérdida de consistencia. Como los datos deben ser sobrescritos por completo, es tentador usar una operación de "eliminar e insertar" (*drop-and-insert*). Sin embargo, si un consumidor consulta los datos mientras la ingesta está en proceso, podría procesar datos parciales o no ver ningún dato si la inserción falla. La mitigación más sencilla es usar **transacciones**, que gestionan la visibilidad de los datos automáticamente. Si el almacén de datos no soporta transacciones, se puede usar una única abstracción de exposición de datos (como una **vista de base de datos**) y manipular solo las tablas técnicas subyacentes. Esto permite intercambiar las tablas de forma atómica desde la perspectiva del consumidor.

Ejemplos

- **Sincronización de buckets S3:** Se puede usar un simple comando como `aws s3 sync s3://input-bucket s3://output-bucket --delete` para sincronizar el contenido de dos buckets.

- **Apache Spark y Delta Lake:** Para cargas que requieren más afinamiento, se puede usar Apache Spark. Su naturaleza distribuida permite escalar sin problemas. Al usarlo con un formato de tabla como Delta Lake, se abordan automáticamente los problemas de consistencia gracias a sus capacidades transaccionales y de versionado.

- **Bases de datos sin versionado nativo:** Se puede implementar el patrón usando Apache Airflow y PostgreSQL. Esto requiere tareas dedicadas para la ingesta (escribir en una tabla versionada, ej. `devices_v1`) y la exposición (actualizar una vista para que apunte a la nueva tabla versionada).

Carga Incremental (*Incremental Load*)

La carga completa puede ser costosa para conjuntos de datos que crecen continuamente. Los patrones de carga incremental son más adecuados porque ingieren partes más pequeñas del conjunto de datos con mayor frecuencia.

Patrón: Cargador Incremental (*Incremental Loader*)

Este patrón procesa las nuevas partes de un conjunto de datos.

Problema En el caso de uso de análisis de un blog, la mayoría de los eventos de visita provienen de un *broker* de *streaming* en tiempo real. Sin embargo, algunos todavía son escritos en una base de datos transaccional por productores heredados. Se necesita un proceso de ingesta dedicado para llevar estas visitas a la capa Bronce, pero debido al volumen de datos en continuo crecimiento, el proceso solo debe integrar las visitas añadidas desde la última ejecución.

Solución Un conjunto de datos en continuo crecimiento es una buena condición para usar el patrón *Incremental Loader*. Hay dos implementaciones posibles:

1. **Usar una columna delta:** Se utiliza una columna (típicamente la fecha de ingesta para datos inmutables) para identificar las filas añadidas desde la última ejecución.

2. **Basarse en particiones de tiempo:** El trabajo de ingesta utiliza particiones basadas en tiempo para detectar el nuevo conjunto de registros a ingerir. Esto simplifica el proceso, ya que los datos ya están lógicamente organizados. Para asegurar que una partición esté completa antes de la ingesta, se puede usar el patrón **Readiness Marker**.

Consecuencias

- **Eliminaciones Físicas (*Hard deletes*):** El patrón se complica con datos mutables. Si se usa la columna delta, se pueden identificar las filas actualizadas, pero no las eliminadas, ya que estas desaparecen físicamente de la fuente. Una solución es que el productor utilice **eliminaciones lógicas (*soft deletes*)**, marcando una fila como eliminada mediante una actualización (`UPDATE`) en lugar de una eliminación física (`DELETE`).

- **Tablas de solo inserción (*Insert-Only Tables*):** Otra solución es usar tablas que solo aceptan nuevas filas mediante una operación `INSERT`. La responsabilidad de reconstruir el estado recae en los consumidores.

- **Relleno de datos históricos (*Backfilling*):** Si se necesita reprocesar datos históricos, un pipeline diseñado para cargas incrementales podría enfrentarse a una carga completa, requiriendo más recursos. Esto se puede mitigar **limitando la ventana de ingesta** (por ejemplo, a una hora). Esto permite un mejor control sobre el volumen de datos y la posibilidad de ejecutar múltiples trabajos de *backfilling* en paralelo.

Ejemplos

- **Sincronización de S3 para una partición:** `aws s3 sync s3://input/date=2024-01-01 s3://output/date=2024-01-01 --delete`.

- **DAG en Airflow para carga incremental:** Se puede usar un `FileSensor` de Airflow para esperar a que una nueva partición de datos esté disponible (marcador de preparación) antes de lanzar un trabajo de Spark para procesarla.

- **Trabajo de ingesta con columna delta:** El trabajo de Spark incluye una operación de filtrado adicional sobre la columna delta para obtener solo las filas dentro del rango de tiempo configurado para esa ejecución.

Patrón: Captura de Datos de Cambio (*Change Data Capture - CDC*)

El *Incremental Loader* no funciona para todos los casos de uso. Cuando se necesita una latencia de ingesta menor o soporte nativo para eliminaciones físicas, el CDC es una mejor opción.

Problema Los eventos de visita heredados del patrón anterior tienen una tasa de ingesta demasiado lenta. Los consumidores se quejan del tiempo de espera. Se requiere integrar estos registros transaccionales en el *broker* de *streaming* lo antes posible, capturando cada cambio en la tabla en menos de 30 segundos.

Solución El requisito de latencia hace imposible usar el *Incremental Loader*. El patrón **Change Data Capture (CDC)** es un mejor candidato. Consiste en ingerir continuamente todas las filas modificadas directamente desde el **registro de confirmaciones (*commit log*)** interno de la base de datos. Esto permite un acceso de bajo nivel y más rápido a los registros. El consumidor de CDC transmite estos cambios a un *broker* de *streaming* u otra salida, garantizando una menor latencia y capturando todas las operaciones, incluidas las eliminaciones físicas.

Consecuencias

- **Complejidad:** El patrón CDC requiere habilidades de configuración diferentes. A menudo se necesita ayuda del equipo de operaciones para habilitar el *commit log* en los servidores de bases de datos.

- **Alcance de los datos:** Dependiendo de la implementación, es posible que solo se obtengan los cambios realizados después de iniciar el proceso cliente. Para datos históricos, podría ser necesario combinar CDC con otros patrones.

- **Carga útil (*Payload*):** CDC añade metadatos adicionales a los registros, como el tipo de operación (insertar, actualizar, eliminar), el tiempo de modificación o el tipo de columna.

- **Semántica de los datos:** Este patrón ingiere "datos en reposo" y los convierte en "datos en movimiento". Esto es importante porque los datos en movimiento tienen semánticas de procesamiento diferentes para operaciones que parecen triviales en el mundo de los datos en reposo, como una operación `JOIN`.

Ejemplos

- **Debezium y Kafka Connect:** Debezium es una popular solución de código abierto que utiliza Kafka Connect como puente. La configuración es principalmente declarativa a través de un archivo JSON que define los parámetros de conexión y los esquemas a monitorear.

- **Change Data Feed (CDF) de Delta Lake:** Los formatos nativos del *lakehouse* soportan CDC de una manera más simple. Delta Lake tiene una función incorporada de *Change Data Feed* que se puede habilitar en una tabla. Luego, se pueden leer estos cambios como un *stream*, obteniendo columnas adicionales que describen el cambio.

Replicación

Otra familia de patrones de ingesta de datos es la replicación, cuyo objetivo principal es copiar los datos "tal cual" de una ubicación a otra.

Carga de Datos vs. Replicación

Son conceptos similares, pero la replicación se refiere al movimiento de datos entre el mismo tipo de almacenamiento, preservando idealmente todos sus atributos de metadatos (ej. claves primarias). La carga es más flexible.

Patrón: Replicador de Passthrough (*Passthrough Replicator*)

Problema El proceso de despliegue consta de tres entornos: desarrollo, *staging* y producción. Muchos trabajos utilizan un conjunto de datos de referencia que se carga diariamente en producción desde una API externa no idempotente (puede devolver resultados diferentes para la misma llamada). Para una mejor experiencia de desarrollo, se necesita tener este mismo conjunto de datos en los otros entornos.

Solución Un proveedor de datos no idempotente y la necesidad de consistencia entre entornos es una excelente razón para usar el patrón *Passthrough Replicator*. La implementación se basa en un trabajo de EL (leer y escribir) que copia los datos "tal cual", sin transformaciones que puedan introducir problemas de calidad.

Consecuencias

- **Mantenlo simple:** Para reducir el riesgo de interferencia, se debe usar el trabajo de replicación más simple posible, idealmente un comando de copia nativo de la base de datos.

- **Seguridad y aislamiento:** La comunicación entre entornos es delicada. Se debe implementar la replicación con un enfoque de "empuje" (*push*) en lugar de "tirón" (*pull*), lo que significa que el entorno que posee el conjunto de datos lo copiará a los demás, controlando así el proceso.

- **Datos PII:** Si el conjunto de datos replicado almacena información de identificación personal (PII), se debe utilizar el patrón *Transformation Replicator*.

- **Latencia y Metadatos:** Se debe tener en cuenta la latencia adicional en las implementaciones basadas en infraestructura y la importancia de replicar correctamente los metadatos.

Patrón: Replicador con Transformación (*Transformation Replicator*)

Problema El mismo escenario que el anterior, pero el conjunto de datos replicado contiene datos PII que no son accesibles fuera del entorno de producción. Como resultado,

no se puede usar un simple trabajo de *Passthrough Replicator*.

Solución Se debe implementar el patrón *Transformation Replicator*, que, además de las partes clásicas de lectura y escritura, tiene una capa de transformación intermedia. La transformación consiste en reemplazar los atributos que no deben replicarse (por ejemplo, con un patrón de anonimización) o simplemente eliminarlos.

Consecuencias

- **Riesgo de transformación:** Al escribir lógica personalizada, el riesgo de romper el conjunto de datos es mayor. Se debe seguir aplicando el enfoque de "mantenlo simple". Por ejemplo, en lugar de definir columnas de marca de tiempo como tales, se pueden configurar como cadenas de texto para no preocuparse por transformaciones silenciosas.

- **Desincronización:** Los datos evolucionan continuamente. Para evitar problemas relacionados con la privacidad, se debe confiar en una herramienta de gobernanza de datos, como un catálogo de datos o un contrato de datos, donde se etiqueten los campos sensibles.

Compactación de Datos (*Data Compaction*)

Incluso un conjunto de datos perfecto puede convertirse en un cuello de botella, especialmente cuando crece con el tiempo. Las operaciones relacionadas con metadatos, como listar archivos, pueden tardar más que las propias transformaciones de datos.

Patrón: Compactador (*Compactor*)

La forma más fácil de abordar el problema de un conjunto de datos en crecimiento es reducir la huella de almacenamiento de los archivos subyacentes.

Problema Un *pipeline* de ingesta de datos en tiempo real sincroniza eventos desde un *broker* de *streaming* a un almacenamiento de objetos. Después de tres meses, todos los trabajos por lotes sufren del problema de sobrecarga de metadatos debido a la gran cantidad de archivos pequeños que componen el conjunto de datos. Pasan el 70% de su tiempo de ejecución listando archivos y solo el 30% procesando los datos.

Solución Tener muchos archivos pequeños es un problema conocido en la ingeniería de datos. La solución natural es almacenar menos archivos. El patrón *Compactor* aborda el problema combinando múltiples archivos más pequeños en otros más grandes, reduciendo así la sobrecarga general de E/S en la lectura. La implementación varía según la tecnología (por ejemplo, el comando `OPTIMIZE` en Delta Lake).

Consecuencias

- **Compensaciones de costo vs. rendimiento:** El trabajo de compactación es un trabajo de procesamiento de datos regular que puede ser intensivo en cómputo. Se debe elegir una estrategia y aceptar que puede no ser perfecta tanto desde la perspectiva del costo como del rendimiento.

- **Consistencia:** La compactación es mucho más simple y segura de implementar en formatos de archivo de tabla modernos con propiedades ACID (como Delta Lake) que en formatos de archivo crudos (como JSON y CSV).

- **Limpieza:** El trabajo de compactación puede preservar los archivos de origen. Por lo tanto, los archivos pequeños seguirán allí. Se debe complementar con un trabajo de limpieza (como el comando `VACUUM`) para reclamar el espacio ocupado.

Preparación de Datos (*Data Readiness*)

La siguiente pregunta problemática que seguramente te harás es: "¿Cuándo debo iniciar el proceso de ingesta?"

Patrón: Marcador de Preparación (*Readiness Marker*)

El *Readiness Marker* es un patrón que ayuda a activar el proceso de ingesta en el momento más apropiado. Su objetivo es garantizar la ingesta del conjunto de datos completo.

Problema Se ejecuta un trabajo por lotes cada hora que prepara datos en la capa Plata. Otros equipos dependen de él para generar modelos de ML y paneles de BI. Pero a menudo se quejan de conjuntos de datos incompletos y han pedido un mecanismo que les notifique cuándo pueden empezar a consumir los datos.

Solución El problema es visible en *pipelines* lógicamente dependientes pero físicamente aislados. No es posible que tu trabajo active directamente los *pipelines* posteriores. En su lugar, puedes marcar tu conjunto de datos como listo para procesar con el patrón *Readiness Marker*.

- La primera implementación utiliza un evento para señalar la completitud del conjunto de datos, a menudo implementado con un **archivo de bandera** (por ejemplo, `_SUCCESS` en Apache Spark).

- Una implementación diferente se aplica a las fuentes de datos particionadas. Si se generan datos para particiones basadas en tiempo, el *Readiness Marker* puede ser convencional. Por ejemplo, si un trabajo se ejecuta cada hora, la ejecución de las 11:00 escribe en la partición 11. Un consumidor que quiera procesar la partición 10, simplemente debe esperar a que el trabajo trabaje en la partición 11.

Consecuencias

- **Falta de cumplimiento:** No hay una manera fácil de hacer cumplir la preparación convencional. Un consumidor puede empezar a consumir el conjunto de datos mientras se está generando.

- **Fiabilidad para datos tardíos:** Si las particiones se basan en el tiempo del evento, la implementación basada en particiones sufrirá problemas de datos tardíos.

Impulsado por Eventos (*Event Driven*)

En escenarios ideales, los nuevos conjuntos de datos están disponibles en un horario regular. En escenarios menos ideales, es difícil predecir la frecuencia de llegada. En consecuencia, se debe cambiar la mentalidad de una ingesta estática a una ingesta impulsada por eventos.

Patrón: Disparador Externo (*External Trigger*)

El patrón *Readiness Marker* se basa en la semántica de "extracción" (*pull*), en la que el consumidor es responsable de verificar si hay nuevos datos. La naturaleza impulsada por eventos de un conjunto de datos favorece la semántica de "empuje" (*push*), en la que el productor se encarga de notificar a los consumidores sobre la disponibilidad de los datos.

Problema El equipo de *backend* de tu organización lanza nuevas características como máximo una vez a la semana. Cada lanzamiento enriquece tus conjuntos de datos de referencia. Hasta ahora, el trabajo de actualización para este conjunto de datos se ha programado una vez al día, recargando datos incluso cuando no había cambios. Para reducir costos, se quiere cambiar el mecanismo de programación y ejecutar el *pipeline* solo cuando hay algo nuevo que procesar.

Solución Los datos de generación impredecible a menudo son causados por la naturaleza impulsada por eventos de los datos. Un mejor enfoque es abordar este problema con un patrón de **Disparador Externo** impulsado por eventos. El patrón consta de tres acciones principales:

1. **Suscripción a un canal de notificación.**

2. **Reacción a las notificaciones.**

3. **Activación del *pipeline* de ingesta.**

Consecuencias

- **Empuje vs. Extracción:** Una mejor alternativa para este patrón es el disparador basado en empuje, donde la fuente de datos informa al punto final sobre los nuevos mensajes.

- **Contexto de ejecución:** Es importante enriquecer la llamada de activación con cualquier información de metadatos apropiada, incluida la versión del trabajo de activación, la envoltura de la notificación, el tiempo de procesamiento y el tiempo del evento.

- **Gestión de errores:** Los eventos son los elementos clave aquí, y sin ellos, no podrás activar ningún trabajo. Por esa razón, debes diseñar el disparador para el fracaso con el objetivo en mente de mantener los eventos pase lo que pase.

Resumen

La ingesta de datos es una operación aparentemente simple que conlleva desafíos significativos. Sin un *Readiness Marker*, se pueden ingerir datos incompletos. Sin el *Compactor*, un *lakehouse* ilimitado puede convertirse en un cuello de botella de rendimiento.

Los patrones discutidos no se reservan solo para la puerta de entrada de tu sistema o para simples *pipelines* de EL. Son excelentes candidatos para el paso de extracción en los *pipelines* de ETL y ELT, por lo que su importancia no debe subestimarse.

El libro en este capítulo hace referencia a las figuras 2-1 y 2-2, pero en el PDF que teníamos en el material de estudio las mismas no aparecían. No sé si es un bug o qué, pero acá no están por eso.