



InDatiOn

Manual de usuario

**Control de Cambios**

|  |  |  |
| --- | --- | --- |
| **Histórico de Versiones** | | |
| **Versión** | **Fecha** | **Resumen de los cambios producidos** |
| 1.0 | 09/12/2020 | Primera versión: Manual unificado |
| 9.3 | 06/05/2024 | Primera versión con Unity Catalog |
|  |  |  |
|  |  |  |

|  |  |
| --- | --- |
| **Responsabilidad** | **Nombre** |
| Preparación |  |
| Revisión |  |
| Aprobación |  |

**INDICE**

[1 Introducción 4](#_Toc1069848159)

[1.1 Capas 5](#_Toc1400990897)

[1.1.1 Landing 5](#_Toc546438381)

[1.1.2 Bronze 6](#_Toc519799972)

[1.1.3 Silver 7](#_Toc1508650493)

[1.1.4 Capa Gold 8](#_Toc1667304880)

[1.2 Ejecución 9](#_Toc1236116731)

[1.2.1 Fichero de configuración yml 9](#_Toc1954425353)

[1.3 Seguridad 10](#_Toc935453623)

[1.3.1 Identidad 10](#_Toc393262429)

[1.3.2 Lectura de secretos 11](#_Toc561976578)

[2 Motor de Ingesta - Configuracion 11](#_Toc927416071)

[2.1 Flujo de ingesta 12](#_Toc200656344)

[2.1.1 Ingesta batch de ficheros 12](#_Toc889015691)

[2.1.2 Ingesta streaming 18](#_Toc960625826)

[2.2 Metadata 19](#_Toc672442858)

[2.2.1 Persistencia sobre ficheros 19](#_Toc1609260607)

[2.3 Ejecución 37](#_Toc1700028615)

[2.3.1 Parámetros de entrada 37](#_Toc873083358)

[3 Motor de metadatos Gold - Configuración. 37](#_Toc1650387599)

[3.1 Procesamiento de datos 38](#_Toc1431618480)

[3.2 Metadata 38](#_Toc1535381288)

[3.2.1 Conexiones 38](#_Toc1056697476)

[3.3 Transformaciones 39](#_Toc828249777)

[3.3.1 Definición de transformaciones 40](#_Toc545392310)

[3.3.2 Bloque “sqlTransformation” 51](#_Toc1942931758)

[3.4 Actividades 51](#_Toc876330512)

[3.4.1 Parámetros de actividad 51](#_Toc1260098331)

[3.4.2 Ejemplo de actividad 51](#_Toc460097022)

[3.5 Ejecución 51](#_Toc410422555)

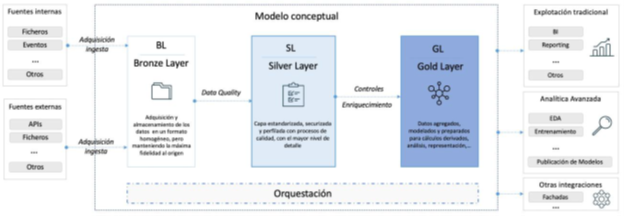
[3.5.1 Parámetros de entrada 51](#_Toc1390725522)

# Introducción

Este documento presenta el modelo de configuración base del motor de ingesta y de la definición de los metadatos requeridos para la creación de ingestas de diferentes tipos de origen. Además, como parte del flujo estándar de datos en la nueva plataforma analítica de datos, se presenta el modelo de configuración base del motor de metadatos gold y de la definición de los ficheros requeridos para la creación de transformaciones de datos.

## Capas

Las capas intervenidas por el motor de ingesta son: *landing*, *bronze, silver y gold*.



### Landing

La capa *landing* funciona como depósito de la información que llega desde los orígenes y se quiere ingestar en el Data Lake.

Se encuentra separada del resto de capas del Data Lake para evitar que a este llegue información que no está gobernada.

La estructura de la capa *landing* (usada para ingesta desde ficheros) se ajustará a la siguiente segmentación:

|  |  |
| --- | --- |
|  | Landing (Ficheros) |
| **Almacenamiento** | Azure Datalake Storage Gen 2 |
| **Contenedor** | landing |
| **Sub-capa** | pending, unknown, schema-mismatch, corrupted, invalid y streaming |
| **Formato de datos** | Según formato origen (binario o texto) |
| **Estructura de la información** | <origen>/<dataset>/<yyyy>/<mm>/<dd> (subcapas schema-mismatch e invalid) |
| **Ordenación de la información** | Fecha de recepción |

Los ficheros llegarán a la capa de *landing* por los mecanismos internos correspondientes. La carpeta que recepciona esta información será la subcarpeta *pending* dentro de la cual se contará con una relación de carpetas, correspondiendo cada una de ellas a un origen dado de alta en el metadata.

Si la ingesta a la capa *bronze* se ha podido realizar correctamente, el fichero será eliminado de la carpeta *landing/pending/<origen>*.

Si el fichero no tiene metadata asociado en el motor de ingesta, el fichero será movido de la carpeta *landing/pending/<origen>* a la carpeta *landing/unknown/<origen>*.

Si el fichero está corrupto y no puede leerse, el fichero será movido de la carpeta *landing/pending/<origen>* a la carpeta *landing/corrupted/<origen>*.

Si la estructura del fichero no se corresponde con la del esquema, el fichero será movido de la carpeta *landing/pending/<origen>* a la carpeta *landing/schema-mismatch/<origen>/<dataset>/<yyyy>/<mm>/<dd>*.

Si el fichero no pasa las reglas de calidad, el fichero será movido de la carpeta *landing/pending/<origen>* a la carpeta *landing/invalid/<origen>/<dataset>/<yyyy>/<mm>/<dd>*.

La ingesta de topics se apoya en la carpeta *streaming* para persistir los checkpoints de Spark para realizar ingestas incrementales. Adicionalmente cada ingesta incremental se persistirá temporalmente en esta carpeta en formato Avro para posteriormente realizar las validaciones necesarias para su ingesta en *bronze* y *silver*.

La estructura de la carpeta streaming será *landing/streaming/<origen>/<dataset>/checkpoint y* será *landing/streaming/<origen>/<dataset>/batch/<timestamp>* para almacenar los checkpoint y los ficheros temporales respectivamente.

### Bronze

La capa *bronze* funciona como repositorio de información homogéneo de todos los datos ingestados. El formato de persistencia se mantiene, siempre que sea posible, con máxima fidelidad al formato original.

La estructura de la capa bronze se ajustará a la siguiente segmentación:

|  |  |
| --- | --- |
|  | Bronze |
| **Almacenamiento** | Azure Datalake Storage Gen 2 |
| **Contenedor** | datalake |
| **Sub-capa** | - |
| **Formato de datos** | Según formato origen (binario o texto) comprimido |
| **Estructura de la información** | <origen>/<dataset>/<yyyy>/<mm>/<dd> |
| **Ordenación de la información** | Fecha de fichero |

La información almacenada en la capa *bronze* ha pasado la validación de estructura definida en el metadata para el dataset ingestado.

El motor de ingesta persiste en la capa *bronze* una copia de la información recibida. Dentro de la carpeta se contará con una relación de subcarpetas, correspondiendo cada una de ellas a un origen y dataset (*bronze/<origen>/<dataset>/<yyyy>/<mm>/<dd>*). La información de esta capa contiene una copia binaria del fichero que llegó a *landing/pending* para el caso de ingestas batch o un fichero *avro* con el conjunto de registros que se leyeron del topic en la ingesta incremental en streaming.

### Silver

La capa *silver* es el almacén final del proceso de ingesta llevado a cabo en el motor. La información persistida se encuentra estandarizada y perfilada con procesos de calidad.

La estructura de la capa *silver* se ajustará a la siguiente segmentación:

|  |  |
| --- | --- |
|  | Silver |
| **Almacenamiento** | Azure Datalake Storage Gen 2 |
| **Contenedor** | datalake |
| **Sub-capa** | - |
| **Formato de datos** | Tablas delta |
| **Clasificación de la información** | Agrupación funcional |
| **Ordenación de la información** | No aplica |
| **Partición** | Parametrizado metadata |

La información persistida en la capa *silver* se almacenará en formato *delta*. Este formato es una evolución del formato parquet con mejoras para la realización, entre otras funcionalidades, de transacciones ACID, versionado de datos, evoluación de esquema.

Los datos ingestados en *silver* han pasado los controles definidos en el metadata para cada uno de los conjuntos de datos. Entre los controles posibles se encuentran:

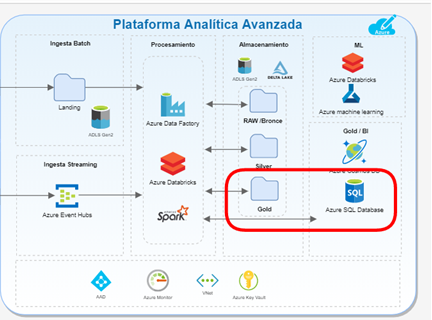
* Transformaciones de campos
* Cumplimiento de esquema (nombres y tipos)

### Capa Gold

El objetivo del motor de metadatos gold es facilitar a los usuarios no expertos en el desarrollo Spark la posibilidad de configurar procesos de transformación de datos que alimenten en la capa Gold de la plataforma con información de negocio orientada a la explotación final de la información.

#### Niveles de almacenamiento Gold

En la arquitectura de la plataforma se han definidos dos niveles de almacenamiento para la capa Gold, siendo el primero de estos la ruta “gold” de Azure Datalake Storage y siendo el segundo de estos el almacenamiento en Azure SQL.



##### Datalake

El nivel de almacenamiento Gold sobre el Datalake está conceptualmente orientado al almacenamiento histórico de los datos de negocio generados para la explotación, con los siguientes objetivos:

* Almacenar la mayor profundidad histórica posible de los datos generados.
* Proporcionar una base de datos detallada para ser explotada por los usuarios avanzados de análisis de datos.
* Un punto de capa “staging” previo a la generación del nivel SQL de explotación masiva de la información.

Debe tenerse en cuenta que este nivel de datos resulta más económico en cuanto al coste de almacenaje, pero menos eficiente desde el punto de vista de la explotación.

Organización

Se ha propuesto como organización dentro de la capa Gold la generación de los datos en función de la unidad organizativa propietaria de la información y la clasificación del tipo de información. Siendo la propuesta de carpetas para la información en Gold la siguiente:

datalake/<empresa>/gold/<clasificación>/<bbdd\_propietario>/<tabla>

La clasificación para todos los datos propuestos en el MVP será siempre “public” y las bases de datos propuestas son:

* GLD\_KPI\_OPERACIONES
* GLD\_KPI\_QUALITY
* GLD\_KPI\_FL
* GLD\_KPI\_RRHH
* GLD\_KPI\_TARGET\_LETTER

##### SQL Database

El nivel Gold de base de datos relacional está orientado al almacenamiento de los datos preparados para su explotación masiva desde herramientas de reporting principalmente. Azure SQL Database proporciona mejor rendimiento a este tipo de consultas, pero tiene un coste de almacenamiento mayor, por lo que se orienta más a almacenar la información más agregada y procesada, con una profundidad histórica menor a la del nivel Datalake.

A nivel de organización, se propone una base de datos Azure SQL para cada unidad organizativa propietaria del dato. Todas ellas estarán vinculadas a un único servidor Azure SQL Server.

## Ejecución

### Fichero de configuración yml

El fichero de configuración YML permite la configuración de las siguientes propiedades para la ejecución del motor de ingesta:

|  |  |  |
| --- | --- | --- |
| Propiedad | Obligatorio | Descripción |
| **metadata.storageAccountName** | Sí | Nombre del recurso de Azure Storage donde se almacenará el metadata en ficheros. |
| **metadata.basePath** | Sí | Ruta base desde la que se estructurará el metadata en ficheros. |
| **metadata.container** | Sí | Nombre del contenedor de Azure Storage que contiene el metadata en ficheros. |
| **databricks.secrets.scope** | Sí | Scope de secretos de Databricks vinculado al Azure Key Vault de las herramientas. |
| **landing.storageAccountName** | Sí | Nombre del recurso Azure Stograge de la capa landing. |
| **landing.basePath** | Sí | Ruta absoluta base de la capa landing. |
| **landing.container** | Sí | Nombre del contenedor de Azure Storage para la capa landing. |
| **landing.pendingDirectory** | Sí | Subdirectorio pending de la capa landing. |
| **landing.unknownDirectoy** | Sí | Subdirectorio ficheros desconocidos capa landing. |
| **landing.invalidDirectory** | Sí | Subdirectorio ficheros no cumplen reglas de calidad. |
| **landing.corruptedDirectory** | Sí | Subdirectorio ficheros corruptos. |
| **landing.schemaMismatchDirectory** | Sí | Subdirectorio ficheros que no cumplen esquema. |
| **landing.streamingDirectory** | Sí | Subdirectorio de landing donde se persistirán los checkpoint de las ingestas de streaming y los fichero temporales Avro. |
| **landing.landingPath** | No | Nombre de la carpeta que contiene el fichero de landing |
| **landing.errorPath** | No | Nombre de la carpeta que contiene los registros con errores de la capa landing |
| **landing.archivePath** | No | Nombre de la carpeta que contiene el fichero landing movido a ruta archive |
| **datalake.storageAccountName** | Sí | Nombre del recurso Azure Stograge del Data Lake. |
| **datalake.basePath** | Sí | Ruta absoluta base del Data Lake. |
| **datalake.container** | Sí | Nombre del contenedor de Azure Storage para el Data Lake. |
| **datalake.bronzePath** | No | Nombre de la carpeta bronze |
| **datalake.silverPath** | No | Nombre de la carpeta silver |
| **datalake.goldPath** | No | Nombre de la carpeta gold |
| **datalake.versionsPath** | No | Nombre de la carpeta versiones |
| **datalake.dataPath** | No | Nombre de la carpeta que contiene la data de bronze |
| **datalake.QAerrorPath** | No | Nombre de la carpeta que contiene los registros erróneos de silver |
| **security.identity.servicePrincipalAppIdKey** | Sí | Application Id del Service Principal utilizado para conectarse a la cuenta de Storage del Data Lake |
| **security.identity.servicePrincialTenantIdKey** | Sí | TenantId del Service Principal utilizado para conectarse a la cuenta de Storage del Data Lake. |
| **security.identity.servicePrincialPasswordKey** | Sí | Key de Azure Key Vault de la que se obtendrá el password del Service Principal. |
| **security.encryption.encryptionType.value** | Sí | Tipo de encriptación. Valores posibles:   * “pseudonymization”: Las columnas que se marquen como “sensitive: true” serán cifradas * “none”: No se aplica ningún tipo de cifrado sobre las columnas de las tablas |
| **security.encryption.masterKey** | No | Nombre del secret de Azure Key Vault que contiene el valor de la clave maestra. Esta clave es la que se usará para cifrar las keys de cifrado generadas en cada proceso de carga donde haya campos sensibles.  **Obligatorio si “security.encryption.encryptionType.value” es de tipo “pseudonymization”** |
| **security.encryption.encryptRawFile** | No | Indica si se desean cifrar los archivos en la capa RAW de aquellos dataset/tablas que contienen algún dato sensible. Valores posibles:   * true: se cifran los ficheros de RAW * false (valor por defecto): no se cifran los ficheros de RAW |
| **security.encryption.hashSalt** | No | Nombre del secret de Azure Key Vault que contiene el valor del salt para el hasheo de los campos sensibles. Al ser el mismo salt siempre, el mismo texto siempre se va a hashear con el mismo resultado (de esta forma, aunque no se vea el valor real, se pueden hacer cálculos sobre la columna).  **Obligatorio si “security.encryption.encryptionType.value” es de tipo “pseudonymization”** |
| **security.encryption.encryptionAlgorithm.algorithm.value** | No | Algoritmo de cifrado que se empleará para cifrar las columnas marcadas como “sensitive: true”. Valores posibles:   * “AES”: cifrado de tipo simétrico   **Obligatorio si “security.encryption.encryptionType.value” es de tipo “pseudonymization”** |
| **security.encryption.encryptionAlgorithm.mode.value** | No | Modo de encriptación empleado en el cifrado simétrico (<https://www.cryptomathic.com/news-events/blog/the-use-of-encryption-modes-with-symmetric-block-ciphers>). Valores posibles:   * “CTR”   **Obligatorio si “security.encryption.encryptionType.value” es de tipo “pseudonymization”** |
| **security.encryption.encryptionAlgorithm.padding** | No | Padding a emplear junto con el modo de encriptación del cifrado simétrico (<https://www.cryptomathic.com/news-events/blog/the-use-of-encryption-modes-with-symmetric-block-ciphers>). Valores posibles:   * “NoPadding”   **Obligatorio si “security.encryption.encryptionType.value” es de tipo “pseudonymization”** |
| **Environment.value** | Sí | Enumerado con los entornos de ejecución compatibles:   * local * databricks |
| **tmpDirectory** | Sí | Ruta donde se almacenarán ficheros temporales generados por el motor. |
| **silverStatistics.statisticsSubPath** | Sí | Ruta donde se encuentra la tabla de statistics en el datalake de silver (ingesta): "silver/interno/applications/indation" |
| **silverStatistics.statisticsDatasetPath** | Sí | Ruta donde se encuentran los metadatos del dataset de la tabla de statistics en el datalake de silver (ingesta): "applications/dataset-indation/dataset-indation.json" |
| **goldStatistics.statisticsSubPath** | Sí | Ruta donde se encuentra la tabla de statistics en el datalake de gold (transformación): "gold/interno/applications/indation" |

## Seguridad

### Identidad

La ejecución del motor se realiza asociada a un *service principal* del directorio activo. Este permite la deficinión del nivel de acceso al motor de acuerdo con el modelo RBAC y ACLs definido por el equipo de seguridad.

Las credenciales del *service principal* se declaran en el fichero de configuración bajo la sección *security.identity*.

security:

  identity:

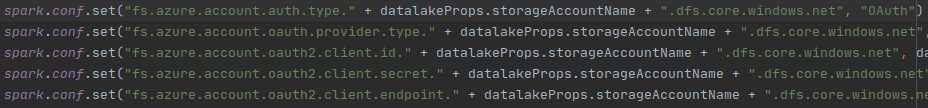
    servicePrincipalAppId: ""

    servicePrincipalTenantId: ""

    servicePrincipalPasswordKey: ""

La clave *security.identity.servicePrincipalPasswordKey* hace referencia al nombre del secreto de Azure Key Vault que contiene el password del *service principal*.

Las credenciales deben incluirse en el contexto Spark por cada uno de los Azure Data Lake Storage utilizandos en el motor. De esta manera Spark es capa de leer rutas que comiencen por abfss:// (prefijo de ADLS)



### Lectura de secretos

La lectura de secretos de Azure Key Vault necesarios para la ejecución del motor, se realiza desde el propio API de Databricks. Para ello es necesario vincular la instancia de Databricks con la instancia de Azure Key Vault.

Este paso debe realizarse desde la interfaz de Databricks bajo la siguiente url:

https:// <workspaceid>.azuredatabricks.net/?o=<workspaceid>**#secrets/createScope**

Es importante que el Scope Name coincida con el nombre del Azure Key Vault para que sea más fácil identificarlo. Databricks no tiene páginas de consulta de los scopes creados y no podremos volver a consultar qué nombre se le dio salvo que se use el databricks-cli.

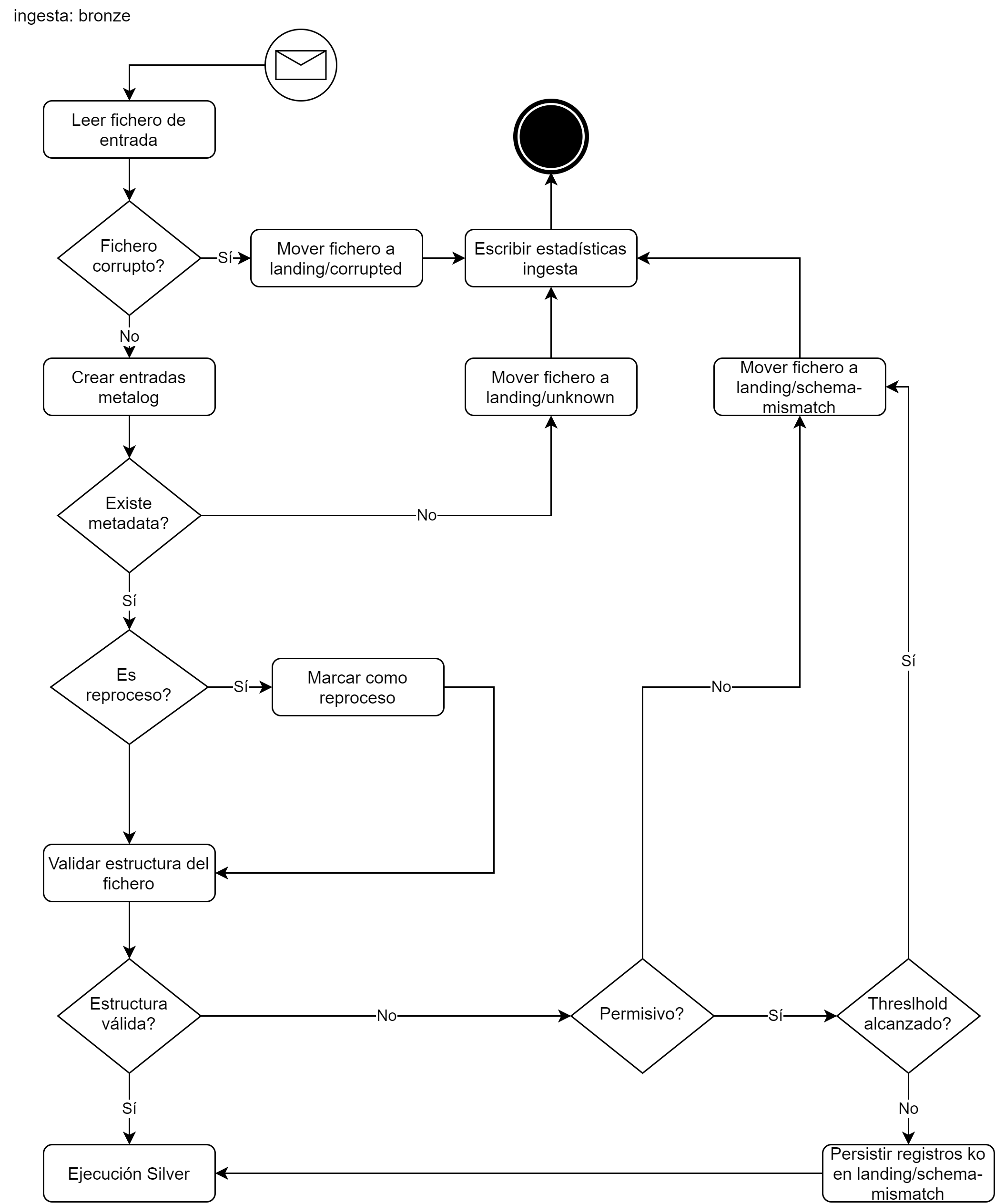
Las propiedades del Azure Key Vault las podemos consultar desde la página de propiedades del mismo.

# Motor de Ingesta - Configuracion

## Flujo de ingesta

### Ingesta batch de ficheros

#### Ingesta batch en capa Bronze



El flujo de ingesta en *bronze* comienza con un evento de llegada de un fichero a la capa landing. El orquestador, ante la llegada de dicho evento, ejecuta el motor de ingesta, pasando por parámetro el path del fichero que acaba de llegar a la capa *landing*.

El motor de ingesta recibe el path del fichero, leerá el fichero y comenzará el flujo de procesamiento necesario para la ingesta en esta capa.

Nada más empezar creará la entrada el metadata de ejecución para registrar las estadísticas de ingesta. En este punto se almacenará el instante de comienzo de la ingesta.

Para un fichero recibido, se comprobará si existe metadata asociado al mismo. La comprobación consiste en buscar en el metadata un patrón de fichero que encaje con el fichero recibido. Los patrones de ficheros tienen un parte fija y una parte variable “<yyyy><mm><dd>\_FICHERO.ext”. Las partículas variables se denotan entre <>. Así para el ejemplo anterior, recibido un fichero de nombre 20200612\_FICHERO.ext encajaría con el patrón anterior y por tanto se catalogaría como reconocido por el metadata. Este tipo de búsqueda obliga a que no puedan repetirse patrones de ficheros para datasets distintos.

En caso de que no se encuentre metadata, se loguea error, y se mueve el fichero de *landing/pending* a *landing/unknown* terminando el proceso de ingesta.

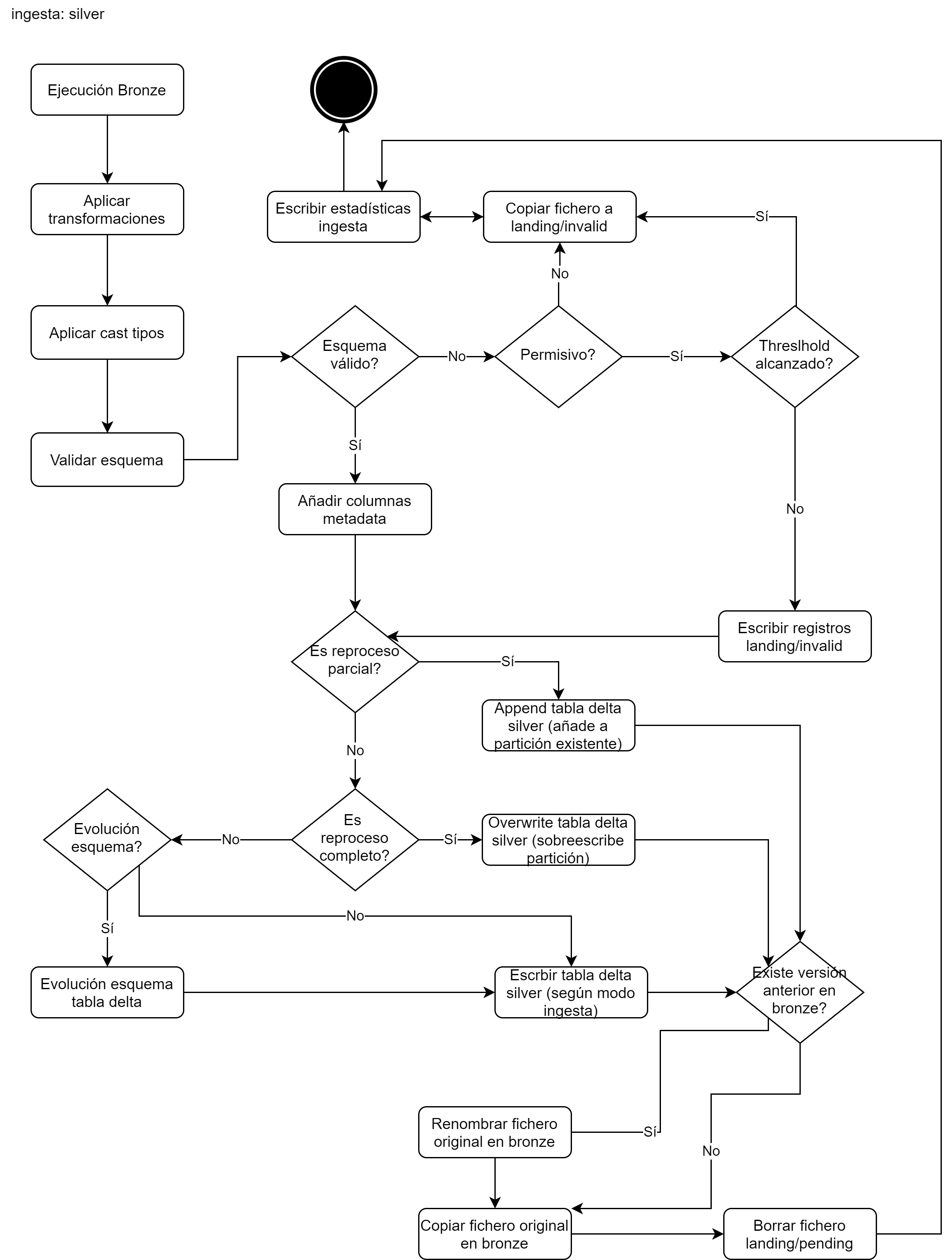
Si existe metadata, se comprueba si se trata de una ingesta de un reproceso parcial o completo. Los reprocesos parciales se identifican cuando el nombre del fichero recibido termina con *“.fix”* mientras que para detectar si se trata de un reproceso completo, se comprueba en el metadata de ejecución si un fichero con el mismo nombre se ha ingestado previamente. Si es necesario realizar un reproceso, se almacena en el metadata de ejecución ya que impacta en la manera de realizar la ingesta posterior en la capa silver.

La actividad de validación de estructura consiste en comprobar que los datos del fichero se ajustan a la estructura proporcionada en el esquema almacenado en el metadata del fichero. No se validan tipos ni contenido. La relación de registros que no cumplan la estructura serán marcados como corruptos. Dependiendo del modo de valicación configurado para el fichero, si existen registros corruptos, se produce el siguiente comportamiento:

* Validación ESTRICTA: Si existe algún registro que no responde al esquema, la ingesta falla. Se loguea dicha situación y el fichero completo se mueve a la carpeta *landing/schema-mismatch*.
* Validación PERMISIVA: Si existen registros que no responden al esquema, se comprueba si el número de estos está por debajo del límite permito. Si es superior, la ingesta falla. Se loguea dicha situación y el fichero completo se mueve a la carpeta *landing/schema-mismatch*. Los límites permitidos pueden estar definidos en número absolutos o relativos. Por ejemplo, se puede aceptar un máximo de 100 registros erróneos o aceptar un máximo del 10% de registros del número total que se lean del fichero. Si el número de errores se encuentra por debajo del límite, los registros erróneos se persisten en *landing/schema-mismatch* y se continúa la ingesta con los registros correctos. El formato de salida de los registros erróneos se mantiene al formato de entrada del fichero.

A continuación, se procede a la ejecución de la capa Silver.

#### Ingesta batch en capa Silver



El flujo de ingesta en *silver* comienza tras las validaciones realizadas en la capa *bronze*. Se aplicarán las transformaciones definidas en el metadata. Estas transformaciones persiguen normalizar la información y aplicar procesos de remediación que ayuden a la correcta conversión de tipos y cumplimiento del esquema de los pasos siguientes.

Las transformaciones aplicables en este punto pueden formar parte del código del motor o extenderse de mediante User-Defined Functions (UDFs). Estas funciones deben inyectarse previamente al contexto spark para que el motor pueda referenciarlas. El nombre de la función a utilizar para procesar un campo formará parte de su metadata cuando se posible.

Aplicadas las transformaciones a la información de *bronze*, se convertirán los tipos de las columnas. Este punto se ejecuta de forma posterior a la transformación para obtener mayor éxito en la aplicación del esquema de tipos si hubiera campos o propiedades que no cumplen los requisitos de conversión tal como vienen de origen (remediación).

A continuación, se valida el esquema completo del dataset. Los requistos de validación en esta capa incluyen ya no solo la estructura, sino los tipos de los campos y la obligatoriedad de estar informados. Dependiendo del modo de valicación configurado para el dataset, si existen registros no válidos, se produce el siguiente comportamiento:

* Validación ESTRICTA: Si existe algún registro no válido, la ingesta falla. Se loguea dicha situación y el dataset se persiste en la carpeta *landing/invalid/<origen>/<dataset>/<yyyy>/<mm>/<dd>/<file>.*
* Validación PERMISIVA: Si existen registros no válidos, se comprueba si el número de estos está por debajo del límite permitido. Si es superior, la ingesta falla. Se loguea dicha situación y el fichero completo se mueve a la carpeta *landing/invalid/<origen>/<dataset>/<yyyy>/<mm>/<dd>/<file>*. Los límites permitidos pueden estar definidos en número absolutos o relativos. Por ejemplo, se puede aceptar un máximo de 100 registros erróneos o aceptar un máximo del 10% de registros del número total que se lean del fichero. Si el número de errores se encuentra por debajo del límite, los registros erróneos se persisten en *landing/invalid/<origen>/<dataset>/<yyyy>/<mm>/<dd>/<file>* y se continúa la ingesta con los registros correctos. El formato de salida de los registros erróneos utilizado será *delta* con tipos *string*.

Al dataset de registros válidos se le añaden por defecto determinadas columnas de metadata y que formarán parte de la tabla delta:

* *datalake\_load\_date*: Timestamp de la ingesta.
* *datalake\_ingestion\_uuid*: Identificador únido de la ingesta.
* *<yyyy>, <mm>, <dd>*, etc: Las partículas variables que forman parte del nombre del fichero se introducen como columnas para permitir particionar por ellas.

En este punto el proceso de ingesta realizará la escritura en las tablas delta en las que está estructurada la capa *silver*. La base de datos a la que pertenece la tabla y el nombre de dicha tabla forman parte del metadata del dataset.

Previamente a la escritura de los registros en la tabla se comprueba si es necesario realizar una evolución de esquema de la tabla. Las evoluciones de esquema permitidas consisten en la adición de nuevas columnas. Si se eliminan columnas en el dataset que existían previmente en la tabla, no se eliminan de esta. El cambio de tipos de columnas existentes no está permitido.

El modo de ingesta depende del modo definido en el metadata del dataset y también está afectado por si se trata de un reproceso parcial o completo. Un reproceso parcial es un subconjuto de registros no válidos en una ingesta anterior. Se trata de correcciones que se han realizado (de manera externa) a registros descartados en el modo permisivo. Un reproceso completo es la reingesta completa de un conjunto que ya se ha ingestado en bloque y cuya partición debe sustituirse por completo.

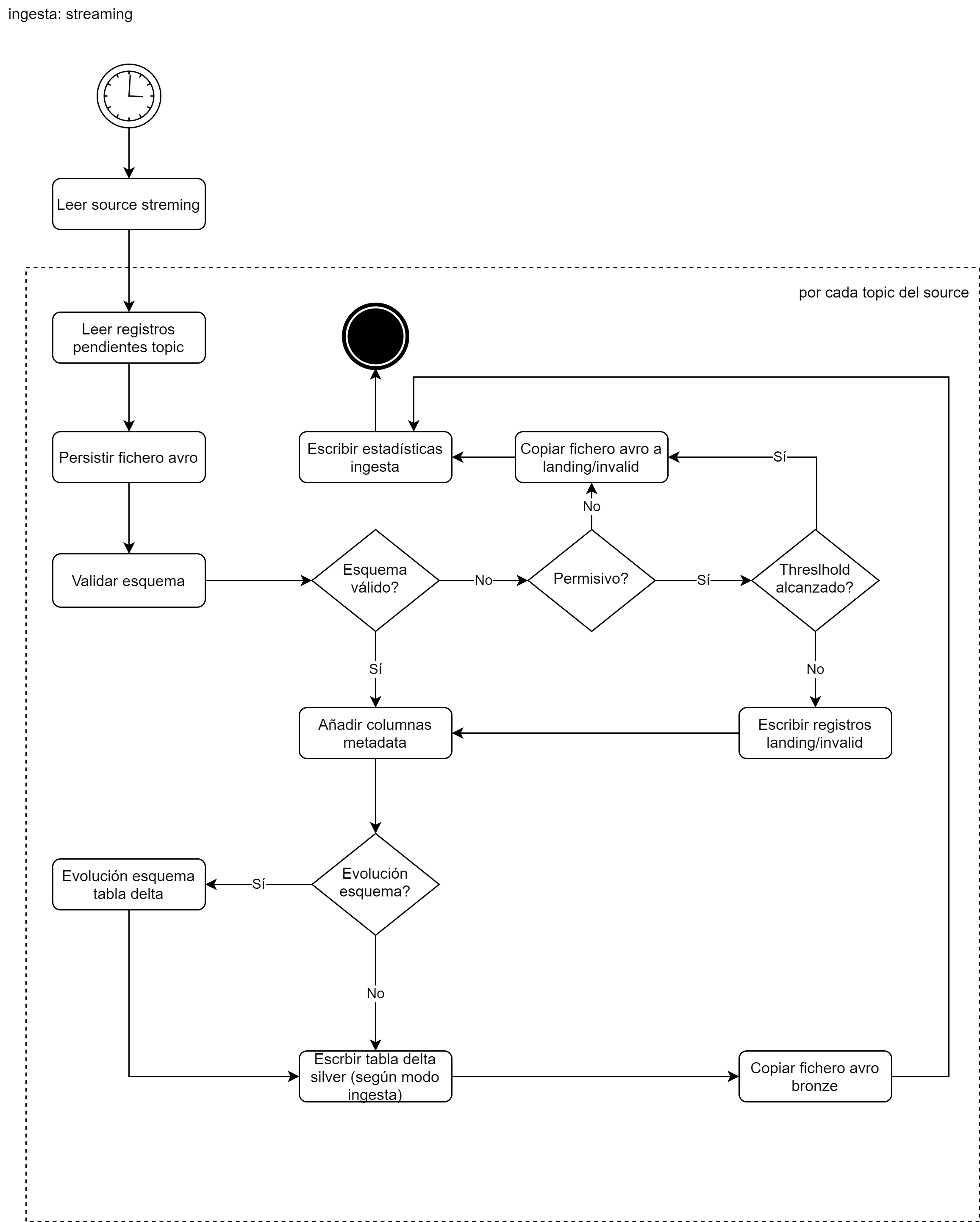
La descripción funcional de los distintos modos de ingesta y su comportamiento particular cuando se ejecutan como reproceso parcial o reproceso completo se definen en la siguiente tabla. Algunos de los modos de ingesta afectan a dos tablas de *silver*, la *principal* (con la fotografía actual de la información) y la *historical*. El modo de escritura varía según la naturaleza de la tabla para toda la combinatoria anterior.

|  |  |  |  |
| --- | --- | --- | --- |
| Modo de ingesta | Batch inicial | Batch reproceso parcial | Batch reproceso completo |
| **Total** | Principal: Reescritura total de la tabla con registros recibidos | Principal: Inserción registros recibidos en la tabla | Principal: Sobreescritura total de la tabla con registros recibidos siempre que la tabla no contenga información más actual |
| Historical: Inserción registros recibidos en nueva partición | Historical: Inserción de los registros recibidos en partición correspondiente | Historical: Sobreescritura de la partición correspondiente con los registros recibidos |
| **Incremetal** | Inserción de los registros recibidos en partición correspondiente | Inserción de los registros recibidos en partición correspondiente | Sobreescritura de la partición correspondiente con los registros recibidos |
| **Ultima situación** | Principal: Inserción de registros nuevos o actualización de los existentes por condición de última situación | Principal: Inserción de registros nuevos o actualización de los existentes por condición de última situación | Principal: Inserción de registros nuevos o actualización de los existentes por condición de última situación |
| Historical: Inserción registros recibidos en nueva partición | Historical: Inserción registros recibidos en nueva partición | Historical: Sobreescritura de la partición correspondiente con los registros recibidos |

Todas las tablas se particionan de acuerdo con las columnas especificadas en el metadata. Una vez la tabla ha sido creada en una ingesta inicial, las columnas de partición no pueden cambiarse y es obligatorio que las tablas estén particionadas por al menos una columna.

El fichero original se copia en la carpeta *bronze/<origen>/<dataset>/<yyyy>/<mm>/<dd>*. Esta carpeta hace la función de repositorio de ficheros ingestados manteniendo la información original. Si en la carpeta ya se encuentra un fichero con el mismo nombre, resultado de una ingesta anterior, dicho fichero será renombrado añadiendo el sufijo “\_vN” a su nombre. N será un secuencial incrementado en cada una de las copias de ficheros en conflicto.

### Ingesta streaming



El flujo de ingesta de los topics de streaming comenzará de forma planificada con un trigger temporal. Por cada uno de los orígenes se programará la cadencia de lanzamiento según el número de eventos que lleguen al topic para alcanzar una solución de compromiso entre la disponibilidad de la información en silver y aprovechamiento eficiente del clúster.

La ingesta recibe por parámetro el source que se quiere ingestar. Este source se corresponde con un bróker Kafka del cual se leeran los topics (datasets) dados de alta en el metadata y se lanzará la ingesta por cada uno de ellos.

La ingesta de un topic consiste en leer los eventos pendientes desde la última ingesta con Spark Structured Streaming y persistirlos de forma temporal en un fichero Avro en la carpeta *streaming* con la siguiente nomenclarura: <yyyy><MM><dd>\_d\_<dataaset-name>\_<tttttttttttttt>\_<m>.bz2.avro <dataset-name>-<yyyy><MM><dd>\_<hh><mm><ss>.bz2.avro. Como se observa en el nombre, se le aplicará el algoritmo de compresión bzip2 al persistirlos. Este fichero Avro se leerá con el API de Dataframe y se procerá a realizar las validaciones correspondientes.

Una vez leído, se compará el esquema del fichero Avro con el que está dado de alta en el metadata. Si hay diferencias se aplicará el modo de validación correspondiente para saber si el fichero se descarta o se procede a su ingesa en silver.

Si el fichero es descartado se moverá a landing/invalid/<origen>/<dataset>/<yyyy>/<mm>/<dd>/<file>. Siendo <yyyy>/<mm>/<dd> el día de la ingesta.

Al dataset de registros válidos se le añaden por defecto determinadas columnas de metadata y que formarán parte de la tabla delta:

* *datalake\_load\_date*: Timestamp de la ingesta.
* *datalake\_ingestion\_uuid*: Identificador únido de la ingesta.
* *<yyyy>, <mm>, <dd>*, etc: Estas columnas se añadirán con el valor correspondiente a la fecha de la ingesta si así está definido en el metadata.

El modo de ingesta depende del modo definido en el metadata del dataset. Los modos de ingesta están descritos en punto 2.1.2 Hay que tener en cuenta que la lógica de ingesta streaming no soportará los reprocesados y siempre se tratará como ingesta inicial (ya que no se reescribirán particiones).

Todas las tablas se particionan de acuerdo con las columnas especificadas en el metadata. Una vez la tabla ha sido creada en una ingesta inicial, las columnas de partición no pueden cambiarse y es obligatorio que las tablas estén particionadas por al menos una columna.

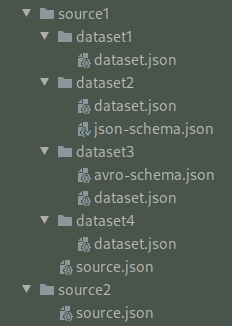
El fichero original se copia en la carpeta *bronze/<origen>/<dataset>/<yyyy>/<mm>/<dd>*. Esta carpeta hace la función de repositorio de ficheros ingestados manteniendo la información original.

## Metadata

El metadata del motor de ingesta está divido en dos secciones diferenciadas: una encargada de informar el catálogo de datasets que pueden ser ingestados en el Data Lake; y otra encargada de mantener la relación de ingestas realizadas.

### Persistencia sobre ficheros

La representación del metadatada sobre ficheros se realiza sobre ficheros json. La estructura se divide en carpetas por source y, dentro de ellas, por dataset.



El almacenamiento del metadata puede configurarse en el fichero de yaml de configuración bajo las propiedades:

metadata:

  storageAccountName: ""

  basePath: ""

  container: ""

#### Source

Los orígenes deben empezar por “source-“ y el nombre del source bajo la siguiente estructura: {sourceFolder}/source-{sourceName}.json y estarán definidos en ficheros json que tendrán diferente estructura en función del tipo de origen. Los tipos de orígenes posibles son “directory”, “jdbc”, “kafka”, y “api”. Dentro de cada uno de ellos, los parámetros a configurar son los siguientes:

Directory:

Los orígnes de tipo directory incluyen únicamente los parámetros obligatorios.

* **name**: nombre del origen de datos que será referenciado desde los datasets.
* **description**: se trata de una descripción libre del origen.
* **typ**: es el tipo de fuente, que se establece como directorio.

{

  "name": "source1",

  "description":"Source 1",

  "typ": {"value":  "directory"}

}

Jdbc:

Los orígenes de tipo Jdbc establecen, además de los tres parámetros obligatorios indicados en el bloque anterior, el bloque jdbcConnection. Este bloque define las características de conexión, la cual depende del tipo de origen (Oracle o SQL Server) y del tipo de autenticación (Usuario/password o SPN). Los parámetros de este bloque son:

* **driver**: se establece el driver jdbc que se va a usar. Actualmente sólo hay dos drivers disponibles y son los que establecen el tipo de bbdd de origen:
  + Oracle: “Oracle.jdbc.driver.OracleDriver”
  + SQL Server y Azure SQL: “com.microsoft.sqlserver.jdbc.SQLServerDriver”
* **host**: dirección del servidor de la instancia de base de datos (IP o nombre DNS)
* **port**: puerto de conexión a la base de datos
* **authenticationType**: tipo de autenticación que se usará para conectar con la base de datos:
  + “user-password”: indica que se autenticará usando un usuario y una contraseña.
  + “service-principal”: indica que se conectará utilizando la entidad de servicio de Azure Active Directory propia del motor de ingesta. Esta autenticación sólo es posible en Azure SQL.
* **sid**: SID del listener de Oracle correspondiente a la instancia de Oracle a la que conectar. Este parámetro es obligatorio cuando el tipo de conexión es Oracle.
* **database**: este parámetro establece el nombre de la base de datos SQL Server o Azure SQL a la que hay que conectar. Este parámetro es obligatorio en el tipo de conexión SQL Server.
* **jdbcUrlKey**: este campo opcional permite especificar la clave de Azure Key Vault donde se encuentra la cadena de conexión. Este campo sustituye la necesidad de los campos host, port, sid y database.
* **userKey**: este parámetro establece el nombre del secreto de Azure Key Vault en el que hay que buscar el nombre de usuario para la conexión. Es obligatorio cuando se establece el tipo de autenticación “user-password”.
* **passwordKey**: este parámetro establece el nombre del secreto de Azure Key Vault en el que hay que buscar la contraseña de usuario para la conexión con la base de datos. Es obligatorio cuando se establece el tipo de autenticación “user-password”.

*Ejemplo Oracle*

{  
 "name": "source-oracle",  
 "description":"Source Oracle",  
 "typ": {"value": "jdbc"},  
 "jdbcConnection": {  
 "driver": {  
 "value": "oracle.jdbc.driver.OracleDriver"  
 },  
 "host": "hostOracle",  
 "port": "1521",  
 "sid": "orcl",  
 "authenticationType": {  
 "value": "user-password"  
 },  
 "userKey": "user\_key",  
 "passwordKey": "password\_key"  
 }  
}

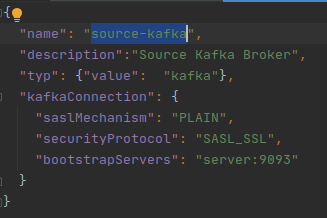
*Ejemplo SQL Server / Azure SQL*

{  
 "name": "source-sqlserver",  
 "description":"Source Sql Server",  
 "typ": {"value": "jdbc"},  
 "jdbcConnection": {  
 "driver": {  
 "value": "com.microsoft.sqlserver.jdbc.SQLServerDriver"  
 },  
 "host": "host-sqlserver",  
 "port": "1433",  
 "database": "databaseName",  
 "authenticationType": {  
 "value": "service-principal"  
 }

}  
}

Kafka:

Los orígenes de tipo Kafka cuenta con unas propiedades que describen la conexión a utilizar:



API:

Los orígenes de tipo Api establecen, además de los tres parámetros obligatorios indicados en el bloque anterior, el bloque apiConnection. Este bloque define las características de conexión, la cual depende del tipo de autenticación (Sin autenticación o Usuario/password). Los parámetros de este bloque son:

* **authenticationType**: tipo de autenticación que se usará para la llamada a la API:
  + “none”: indica que no se necesita autenticación.
  + “basic”: indica que se autenticará con la opción “Basic”, usando un usuario y una contraseña.
* **url**: URL base de la API.
* **userKey**: este parámetro establece el nombre del secreto de Azure Key Vault en el que hay que buscar el nombre de usuario para la conexión a la API. Es obligatorio cuando se establece el tipo de autenticación “basic.
* **passwordKey**: este parámetro establece el nombre del secreto de Azure Key Vault en el que hay que buscar la contraseña de usuario para la conexión a la API. Es obligatorio cuando se establece el tipo de autenticación “basic”.

**Ejemplo**

{  
 "name": "source-api",  
 "description":"Source API",  
 "typ": {"value": "api"},  
 "apiConnection": {  
 "authenticationType": {  
 "value": "basic"  
 },  
 "url": "http://services.odata.org/V4/TripPinService",  
 "userKey": "user\_key",  
 "passwordKey": "password\_key"  
 }  
}

#### Dataset

El nombre de los dataset deben empezar por “dataset-“ y contener el nombre de la carpeta source a continuación: {sourceFolder}/dataset-{sourceFolder}-{datasetName}/ dataset-{sourceFolder}-{datasetName}.json y estarán definidos en ficheros json que tendrán la siguiente estructura:

|  |  |  |
| --- | --- | --- |
| Propiedad | Obligatorio | Descripción |
| **name** | Sí | Nombre del dataset |
| **description** | Sí | Descripción libre del dataset |
| **sourceName** | Sí | Indica el nombre de la fuente que debe utilizar este dataset para acceder a la información. Debe coincidir con el atributo “name” del json de fuente que se ha definido. |
| **typ.value** | Sí | Tipo de dataset. Los valores posibles son:   * “file”: para ingestas de tipo fichero * “table”: para ingestas de tipo jdbc * “topic”: para ingestas de tipo streaming * “api”: para ingestas de tipo llamada a API |
| **version** | Sí | Especifica la versión del dataset. Esto permite tener varias versiones del mismo dataset con inicios de validez diferentes. El motor elige el dataset con la última versión. |
| **enabled** | Sí | Indica si el dataset está operativo. Los valores posibles son “true” o “false” |
| **effectiveDate** | No | Espeficia la fecha de inicio de validez del dataset. Se define en formato “yyyy-MM-dd hh:mm:ss” |
| **ingestionMode** | Sí | Indica el modo de ingesta del dataset. Los valores posibles son:   * “full\_snapshot”: Carga completa * “incremental”: Carga en modo añadir registros nuevos. * “last\_changes”: Añade nuevos registros y actualiza los modificados. |
| **validationMode** | Sí | Especifica el modo de validación de los registros que se leen de un fichero de texto. Los valores posibles son:   * “fail\_fast”: falla en cuanto un registro del fichero no cumple con el esquema previsto. * “permissive”: permite un umbral de filas incorrectas a partir de las que genera error de carga. |
| **createDatabase** | Sí | Espefica si se desea que la base de datos de destino se cree en el metastore de Databricks. Los valores pueden ser “true” o “false”. |
| **database** | Sí | Nombre de la base de datos de destino del metastore de Databricks que alojará la tabla donde se van a cargar los datos. |
| **table** | Sí | Nombre de la tabla de destino de los datos dentro del metastore de Databricks. |
| **partitionBy** | Sí | Especifica la lista de campos del dataset (separados por “/”) para los que se va a particionar la información. |
| **allowPartitionChange** | No | Este campo opcional se utiliza para casos muy concretos en las cargas de tipo lastChanges. Si se establece explícitamente como false, el motor contempla todas las columnas de partición como parte de la primary key, de modo que la búsqueda de los registros con cambios utiliza la optimización de búsqueda por partición. |
| **permissiveThresholdType** | Sí | En el caso de validationMode = “permissive”, especifica si el umbral que se va a establecer es de tipo absoluto o porcentual:   * “absolute”: el valor de umbral que se defina se referirá a un número específico de filas. * “percentage”: el valor de umbral que se defina especificará el porcentaje de filas que pueden fallar. |
| **permisiveThreshold** | Sí | Para validaciones de tipo permissive, especifica el valor del umbral a partir del que se genera error de carga. |
| **fileInput** | No | Define el bloque de atributos para ingestas de tipo fichero. |
| **tableInput** | No | Define el bloque de atributos para ingestas de tipo tabla de base de datos. |
| **kafkaInput** | No | Define el bloque de atributos para ingestas de tipo streaming. |
| **apiInput** | No | Define el bloque de atributos para ingestas de tipo API |
| **schemaDefinition** | Sí | Especifica el tipo de definición de esquema esperado. Los valores posibles son:   * “json-columns” * “json-schema” * “avro-schema” |
| **schemaFile** | No | En los casos de tipo json-schema y avro-schema, especifica la ruta relativa del fichero que contiene la definición de estos esquemas. |
| **schemaColumns** | No | En el caso “json-columns”, es el bloque que define el esquema esperado y se define en el propio cuerpo del json del dataset. |
| **qualityRules** | No | Define las reglas de calidad a aplicar en la capa silver |

##### FileInput

Si se trata de un dataset procedente de un fichero, se debe completar obligatoriamente su propiedad “*fileInput*”. Los atributos de esta propiedad son los siguientes

|  |  |  |
| --- | --- | --- |
| Propiedad | Obligatorio | Descripción |
| **format.value** | Sí | Tipo de fichero. Puede ser uno de los siguientes valores:   * csv * json * text * fixed (fichero de columnas por posición) * parquet * orc * avro * xls (ficheros xls o xlsx) |
| **filePattern** | Sí | Patrón del nombre del fichero. Las partes variables (fechas, etc) pueden definirse con partículas entre **<>**. Dichas partes variables pueden emplearse como columnas para el partitionBy del dataset. |
| **csv** | No | Propiedades que pueden trasladarse al reader de Spark para csv. En el caso de “*header*” es un enumerado con los posibles valores:   * first\_line * without\_header * ignore\_header |
| **json** | No | Propiedades que pueden trasladarse al reader de Spark para json. |
| **fixed** | No | Propiedades que pueden trasladarse al reader de Spark para ficheros de columas por posición. |
| **xls** | No | Propiedades que pueden trasladarse al reader de Spark para xls.  Propiedades:   * Obligatorio: * header: Es un enumerado con los posibles valores:   + first\_line   + without\_header   + ignore\_header * Opcional: * sheet: Selección de la hoja de los datos. * dataRange: Selección de posición de los datos dentro de la hoja. |

##### TableInput

En los casos de una ingesta de tipo tabla, es necesario cumplimentar los atributos de su propiedad “tableInput”. Los atributos de esta propiedad son los siguientes:

|  |  |  |
| --- | --- | --- |
| Propiedad | Obligatorio | Descripción |
| **table** | Sí | Nombre de la tabla o vista de la base de datos que se va a leer. |
| **schema** | No | Nombre del esquema en el que se encuentra la tabla o vista que se va a leer. Si no se define, la tabla se buscará en el esquema por defecto. |
| **query** | No | Si se define este campo, el proceso de injesta jdbc importa el resultado de la consulta sql introducida.  **Obligatorio si “onlyNewData” es true**  Cuando onlyNewData es true, query requiere contener en la cadena de texto lo siguiente: isTimestampMaxSilverValue |
| **fetchsize** | No | Cantidad de registros que spark recupera al leer una tabla de bbdd con cada consulta. Muy útil para lecturas de tablas de Oracle, cuyo valor por defecto es 10.  Ejemplo de valor: “10000” |
| **onlyNewData** | No | En tablas de tipo last\_changes o incremental, indica si se quieren leer únicamente los registros nuevos de la tabla de origen respecto a la tabla en silver (con false, que es el valor por defecto, se lee la tabla completa de origen y se filtran los registros posteriormente). Para ello se empleará el campo indicado con isTimestamp=true.  Esta opción puede ser útil para lecturas de tablas muy grandes.  Valor por defecto: false |
| **datePattern** | No | Permite la conversión del valor de la columna date/datetime indicada como isTimestamp=true a cadena de texto según la máscara definida:   * yyyyMMdd HH:mm:ss * MM/dd/yyyy * dd-MM-yyyy HH:mm:ss * Etc   **Obligatorio si “onlyNewData” es true** |

onlyNewData

El atributo **onlyNewData** en tablas de tipo last\_changes o incremental, indica si se quieren leer únicamente los registros nuevos de la tabla de origen respecto a la tabla en silver.

Cuando el motor realiza la lectura de una tabla de tipo last\_changes o incremental, obtiene en primer lugar el valor máximo del campo indicado como **isTimestamp=true** (el cual debe ser de tipo date o datetime) de la tabla destino de silver. Tras esto, hay 2 posibilidades:

* Si **onlyNewData** es false (valor por defecto):
  + Se lee la tabla completa desde origen, para posteriormente filtrar y quedarnos con los registros cuyo valor de la columna **isTimestamp** es mayor al valor obtenido previamente de la tabla destino de silver, es decir, los registros que son nuevos respecto a la carga anterior.
* Si **onlyNewData** es true:
  + Con el valor máximo del campo **isTimestamp**, se usa el patrón de **datePattern** para convertir dicho date o datetime a una cadena de texto. Posteriormente, se obtendrá el valor de la propiedad **query** y se reemplazará la cadena de texto “isTimestampMaxSilverValue” por la fecha obtenida previamente en tipo texto, obteniendo así una query que ejecutar sobre la tabla de origen para obtener solo aquellos registros que cumplan la query, es decir, los registros que son nuevos respecto a la carga anterior

Ejemplo de tableInput:

"tableInput":{  
 "table": "ProductCategory",  
 "schema": "Production",  
 "query": "select \* from Production.ProductCategory where ModifiedDate > to\_timestamp('isTimestampMaxSilverValue', 'yyyy-MM-dd HH:mm:ss.SSS')",  
 "onlyNewData": true,  
 "datePattern": "yyyy-MM-dd HH:mm:ss.SSS"  
},

Con este ejemplo, internamente el motor crea la siguiente query para ejecutarla sobre la tabla de origen (supongamos que la fecha máxima del campo isTimestamp de silver es 2023-05-15 14:00:00.000):

*select \* from Production.ProductCategory where ModifiedDate > to\_timestamp('2023-05-15 14:00:00.000', 'yyyy-MM-dd HH:mm:ss.SSS')*

Importante tener en cuenta que cada tipo de origen de datos tiene un lenguaje SQL diferente, motivo por el cual se debe generar la query manualmente para adaptarla a cada caso.

##### KafkaInput

En caso de tratarse de un dataset procedente de un topic de Kafka, se deble completar su propiedad “*kafkaInput*”:

|  |  |  |
| --- | --- | --- |
| Propiedad | Obligatorio | Descripción |
| **topic** | Sí | Nombre del topic de Kafka |
| **startingOffsets** | No | Punto de comienzo de la ingesta cuando no se ha ingestado previamente (no existe checkpoint). Valores permitidos:   * earliest * latest |
| **jaasConfiguration** | No | Secreto de Key Vault desde donde se leerá el token de autenticación necesario para conexión a Kafka. |

##### ApiInput

En los casos de una ingesta de tipo tabla, es necesario cumplimentar los atributos de su propiedad “apiInput”. Los atributos de esta propiedad son los siguientes:

|  |  |  |
| --- | --- | --- |
| Propiedad | Obligatorio | Descripción |
| **endpoint** | Sí | Nombre del endpoint de la API del que se quiere obtener información |
| **parameters** | No | Cadena de texto con los parámetros necesarios para hacer la consulta a la API |
| **dataIn** | No | Nombre del atributo del JSON de respuesta donde se encuentran los datos. Si no se indica o está vacío, el nivel superior de la respuesta ya se considera como datos |
| **headers** | No | Cabeceras a incluir en la consulta a la API |
| **pagination** | Si | Especifica la manera en la que se va a obtener la información requerida (depende de si la información se obtiene de forma paginada). |

Parameters

En “parameters” no se puede incluir ningún carácter “ “ (espacio), en su lugar debemos usar “+”.

En caso que la carga sea incremental o last changes, en el proceso se obtendrá la fecha máxima del campo isTimestamp de la carga previa de la tabla final, y dicho valor se incluirá en la url de la llamada a la API en la parte de “parameters”. Para ello, en la cadena de texto de “parameters” se debe incluir este texto, el cual será reemplazado por la fecha obtenida anteriormente:

{lastValue}

Ejemplo de los parámetros para una llamada a una API de tipo OData, donde se pueden poner filtros, por ejemplo, uno sobre el campo “FechaCarga”, campo isTimestamp usado para la carga incremental:

"parameters": "$filter=FechaCarga+gt+{lastValue}"

Durante el proceso, esto se traduce a: $filter=FechaCarga+gt+2022-01-01T00:00:00Z

Pagination

Dentro de “pagination” tenemos las siguientes propiedades:

|  |  |  |
| --- | --- | --- |
| Propiedad | Obligatorio | Descripción |
| **typ.value** | Sí | Especifica la manera en la que se va a obtener la información requerida (depende de si la información se obtiene de forma paginada). Valores posibles:   * next\_link: Se hacen varias llamadas a la API, obteniendo en cada llamada X registros, además de la siguiente url donde se hará la llamada para obtener los siguientes registros * offset: Se hacen varias llamadas a la API, obteniendo en cada llamada X registros. En cada llamada, se genera la siguiente url a la que se llamará con top y skip, para obtener los datos siguientes. * none: Se hace una única llamada a la API (no hay paginación de los datos) |
| **top** | No | Nombre del parámetro en la llamada para indicar cuantos registros obtener en la respuesta por cada llamada  **Obligatorio si “typ.value” es de tipo “offset”** |
| **skip** | No | Nombre del parámetro en la llamada para indicar cuantos registros saltarnos de la respuesta  **Obligatorio si “typ.value” es de tipo “offset”** |
| **topSize** | No | Número de registros a obtener en cada llamada a la API  **Obligatorio si “typ.value” es de tipo “offset”** |
| **nextUrl** | No | Nombre de la propiedad en la respuesta de la llamada a la API donde se encuentra la siguiente url donde consultar para obtener los resultados siguientes a los obtenidos en la llamada actual  **Obligatorio si “typ.value” es de tipo “next\_link”** |

##### SchemaColumns

En caso de definición de dataset de tipo “jsonColumns”, se define el esquema esperado del dataset, en formato json, dentro del propio json del dataset. Contempla, mediante listas de columnas, los siguientes atributos posibles para cada columna:

|  |  |  |
| --- | --- | --- |
| Propiedad | Obligatorio | Descripción |
| **name** | Sí | Nombre de la columna. |
| **typ.value** | Sí | Tipo de dato esperado en la columna. Los valores posibles son:   * integer * string * float * long * boolean * double * decimal * date * short * array * struct |
| **description** | No | Descripción libre de la columna |
| **comment** | No | Comentario referente a la column |
| **isPrimaryKey** | No | Especifica si la columna es clave primaria, lo cual es necesario en ingestas de tipo “lastChanges”. Los valores posibles con “true” o “false” |
| **isTimestamp** | No | Indica si es la columna que indica el momento de creación o actualización del registro. Se requiere para cargas de tipo “incremental” y “lastChanges”. |
| **isPartitionable** | No | Si este valor se define como true en una columna de tipo Date, el motor genera por detrás tres columnas con el mismo nombre y el epígrafe “\_yyyy”, “\_mm”, “\_dd” para año, mes y día respectivamente, que, si son utilizadas como clave de partición, quedan persistidas en el dataset de salida. |
| **transformation** | No | Definición de la transformación a aplicar a columnas en orígenes de tipo fichero de texto plano. |
| **alias** | No | Renombrado de la columna |
| **ignorePersistence** | No | Especifica si la columna se quiere mantener o no. En caso de ser true la columna se elimina. Su valor por defecto es false. |
| **arrayInfo** | No | Información sobre los elementos del array  **Obligatorio si “typ.value” es de tipo “array”** |
| **parentName** | No | Nombre de la columna que contiene a la columna con esta propiedad. Se emplea cuando se trata de una columna que pertenece a una columna de tipo “struct” |
| **decimalParameters** | No | Para las columnas de tipo “decimal”, permite indicar:   * precision: cantidad máxima de dígitos del número * scale: cantidad de dígitos del total que pueden ser decimales |
| **sensitive** | No | Especifica si el valor de la columna se debe cifrar o no (porque se trata de un valor sensible). Su valor por defecto es false |

ArrayInfo

Propiedades que puede contener “arrayInfo”:

|  |  |  |
| --- | --- | --- |
| Propiedad | Obligatorio | Descripción |
| **elementType.value** | Sí | Tipo de datos de los elementos contenidos en el array. Valores posibles:   * integer * string * float * long * boolean * double * decimal * date * short * array * struct |
| **transformation** | No | Definición de la transformación a aplicar a columnas en orígenes de tipo fichero de texto plano. |
| **decimalParameters** | No | Para las columnas de tipo “decimal”, permite indicar:   * precision: cantidad máxima de dígitos del número * scale: cantidad de dígitos del total que pueden ser decimales |

Transformaciones

En datasets de tipo fichero de texto plano (cuyo esquema no va implícito en el fichero), las columnas son susceptibles de aplicación de transformaciones antes de su conversión a los tipos esperados. Esta funcionalidad puede usarse, entre otros usos, para aplicar remediación a los datos de origen. Las transformaciones configurables son:

|  |  |  |
| --- | --- | --- |
| Transformación | Parámetros | Descripción |
| **date** | pattern | Permite la conversión a columna de tipo Timestamp según la máscara definida en el campo pattern:   * yyyyMMdd HH:mm:ss * MM/dd/yyyy * dd-MM-yyyy HH:mm:ss * Etc |
| **comma** | - | Permite la eliminación del separador punto como separador de miles y el cambio de coma por punto como separdor decimal. |
| **udf** | function | Permite la aplicación de una udf a la columna. El parámetro function contiene el nombre de la función a aplicar. Para poder aplicar funciones udf, éstas deben ser desarrolladas de manera ad-hoc en el código del motor. |

##### QualityRules

En esta sección se define la información necesaria para aplicar calidad en los datos de Silver. Se distinguen cuatro tipos de reglas: duplicados, nulos, integridad y de expresión. La regla de integridad contiene la información para revisar que todos los valores de ciertas columnas de la tabla origen están contenidos en la tabla maestra. La regla de expresión dispone de los elementos necesarios para comprobar los números de registros que cumplen con una expresión determinada.

|  |  |  |  |
| --- | --- | --- | --- |
| **Campo** | **Tipo** | **Obligatorio** | **Descripción** |
| **uniquenessRule** | UniquenessRules | No | UniquenessRules contiene una lista de regla de unicidad (rules: List[UniquenessRule]) |
| **notNullRule** | List[String] | No | Lista con las columnas para analizar nulos |
| **IntegrityRule** | IntegrityRules | No | IntegrityRules contiene una lista de regla de integridad (rules: List[InterityRule]) |
| **expressionRule** | ExpressionRules | No | ExpressionRules contiene una lista de reglas de expresión (rules: List[ExpressionRule]) |
| **mode** | QualityRulesMode | Sí | Modo en que se realiza la calidad   * Warning: Solo se tendrán en cuenta los registros fallidos por las reglas de estructura y/o formato * Reject: Además de los registros fallidos por las reglas de estructura/formato, también se tendrán en cuenta los registros fallidos por calidad |

**UniquenessRule**

|  |  |  |  |
| --- | --- | --- | --- |
| **Campo** | **Tipo** | **Obligatorio** | **Descripción** |
| **name** | String | Sí | Nombre de la regla de unicidad |
| **columns** | List[String] | Sí | Lista de columnas a aplicar la regla de unicidad |

**IntegrityRule**

|  |  |  |  |
| --- | --- | --- | --- |
| **Campo** | **Tipo** | **Obligatorio** | **Descripción** |
| **name** | String | Sí | Nombre de la regla de integridad |
| **localColumns** | List[String] | Sí | Columnas de la tabla de origen |
| **referencedTable** | String | Sí | Tabla maestra que también ha de incluir la base de datos donde se encuentra. Ejemplo: “bbdd.tabla” |
| **referencedColumns** | List[String] | Sí | Columnas de la tabla maestra a comparar con la tabla de origen (siguiendo el orden) |

**ExpressionRules**

|  |  |  |  |
| --- | --- | --- | --- |
| **Campo** | **Tipo** | **Obligatorio** | **Descripción** |
| **name** | String | Sí | Nombre de la regla de expresión |
| **expr** | String | Sí | Expresión a aplicar |

Ejemplo qualityRules:

"qualityRules": {  
 "mode": { "value": "reject" },  
 "uniquenessRule": {  
 "rules": [  
 {  
 "name": "uniqueRule1",  
 "columns": ["name","country"]  
 }  
 ]  
 },  
 "notNullRule": ["subcountry"],  
 "integrityRule": {  
 "rules": [  
 {  
 "name": "Integrity1",  
 "localColumns": ["country"],  
 "referencedTable": "database1.incr\_worldcities",  
 "referencedColumns": ["country"]  
 }  
 ]},  
 "expressionRule": {  
 "rules": [  
 {  
 "name": "expr1",  
 "expr": "geonameid > 7"  
 }  
 ]  
 }  
}

##### Json-schema

Este tipo de definición de esquema está orientado a la recepción de información semi-estructurada. Por tanto, serán esquemas aplicables a formatos de entrada json. Si bien gracias a su alto nivel de expresividad, pueden utilizarse para definir la estructura de cualquier formato

La definición se ajusta a la propuesta de <http://json-schema.org/>

{

    "$schema": "http://json-schema.org/draft-04/schema#",

    "type": "object",

    "additionalProperties": true,

    "properties": {

        "object": {

            "type": "object",

            "additionalProperties": false,

            "properties": {

                "item1": {

                    "type": "string"

                },

                "item2": {

                    "type": "string"

                }

            },

            "required": [

                "item1",

                "item2"

            ]

        },

        "array": {

            "type": "array",

            "items": {

                "type": "object",

                "additionalProperties": true,

                "properties": {

                    "itemProperty1": {

                        "type": "string"

                    },

                    "itemProperty2": {

                        "type": "number"

                    }

                }

            }

        },

        "structure": {

            "type": "object",

            "additionalProperties": false,

            "properties": {

                "nestedArray": {

                    "type": "array",

                    "items": {

                        "type": "object",

                        "additionalProperties": false,

                        "properties": {

                            "key": {

                                "type": "string",

                                "enum": [

                                    "KEY1",

                                    "KEY2"

                                ]

                            },

                            "value": {

                                "type": "integer"

                            }

                        }

                    }

                }

            }

        },

        "integer": {

            "type": "integer"

        },

        "string": {

            "type": "string"

        },

        "number": {

            "type": "number"

        },

        "float": {

            "type": "float"

        },

        "nullable": {

            "type": [

                "number",

                "null"

            ]

        },

        "boolean": {

            "type": "boolean"

        },

        "additionalProperty": {

            "type": "string"

        }

    },

    "required": [

        "object",

        "array",

        "structure",

        "typeless",

        "integer",

        "string",

        "number",

        "nullable",

        "boolean"

    ]

}

Para la lectura y conversión del esquema a formato spark puede utilizar la librería <https://github.com/zalando-incubator/spark-json-schema>

##### Avro-schema

Este tipo de definición se ajusta a la especificación de esquemas en avro y será aplicables a este tipo de formato de entrada.

La especificación del esquema se encuentra en <https://avro.apache.org/docs/current/spec.html>



#### ActivityLog

Las ejecuciones tendrán como resultado un fichero json con la siguiente estructura:

{

    "dataset":{

       "ingestion\_mode":"full\_snapshot",

       "name":"full-snapshot-worldcities",

       "partition\_by":"yyyy/mm/dd",

       "typ":"file",

       "validation\_mode":"fail\_fast",

       "version":1

    },

    "engine":{

       "name":"ingestion",

       "version":"1.0.0-SNAPSHOT-6"

    },

    "execution":{

       "duration":25.3700008392334,

       "end":"2020-09-22 09:26:24.688",

       "start":"2020-09-22 09:25:59.318"

    },

    "origin":"abfss://landing@storage.dfs.core.windows.net/company/pending/sslsui/20200101\_worldCities.csv.gz",

    "output\_paths":{

       "bronze":"abfss://datalake@storage.dfs.core.windows.net/company/bronze/source1/full-snapshot-worldcities/2020/01/01/20200101\_worldCities.csv.gz",

       "silver\_historical":"abfss://datalake@storage.dfs.core.windows.net/company/silver/public/database1/full\_snapshot\_worldcities\_historical",

       "silver\_principal":"abfss://datalake@storage.dfs.core.windows.net/company/silver/public/database1/full\_snapshot\_worldcities"

    },

    "result":"INGESTED\_WITHOUT\_ERRORS",

    "rows":{

       "bronze\_invalid":0,

       "bronze\_valid":23018,

       "silver\_invalid":0,

       "silver\_valid":23018

    },

    "silver\_persistence":{

       "database":"database1",

       "historical\_current\_version":0,

       "historical\_table":"full\_snapshot\_worldcities\_historical",

       "principal\_current\_version":0,

       "principal\_table":"full\_snapshot\_worldcities"

    },

    "trigger":{

       "id":"6bcebcc4-9211-476a-8c9d-3a95daece126",

       "typ":"adf"

    },

    "uuid":"b77fad55-30b8-4031-a686-bf8253476dd1"

 }

Los activity logs se persistirán en una tabla en silver definina en un Dataset que debe tener el nombre ***indation***.

## Ejecución

### Parámetros de entrada

El motor podrá recibir parámetros de ejecución a traves de la línea de comandos. En el caso de ser lanzados a través del orquestador, estos también podrán ser inyectados por el mismo.

La especificación de los parámetros de entrada seguirá la convención GNU. De esta manera los parámetros son autoexplicativos y es fácil determinar su función.

|  |  |  |  |
| --- | --- | --- | --- |
| Parámetro | Enumerado | Obligatorio | Descripción |
| **--adf-runid** | - | No | Id de ejecución del pipeline de Data Factory. De esta manera podemos vincular la ejecución del motor con la ejecución del pipeline que lo invocó. Se puede obtener de la variable de Data Factory @pipeline().RunId |
| **--ingest-file (DEPRECADO)** | - | No | **DEPRECADO**  Fichero que se quiere procesar en el motor para comandos de ingesta. Desde Data Factory se puede obtener con la siguiente variable @concat(concat(pipeline().parameters.sourceFolder, '/'), pipeline().parameters.sourceFile) |
| **--ingest-dataset** | - | No | Nombre del dataset que se quiere procesar. Los datasets deben guardarse en la carpeta “ingestion” siguiendo la nomenclatura:  {sourceFolder}/dataset-{sourceFolder}-{datasetName}/dataset-{sourceFolder}-{datasetName}.json  Br El nombre del dataset seria únicamente la última parte, sin el .json”:  --ingest-dataset dataset-{sourceFolder}-{datasetName} |
| **--dataset-path** | - | No | Path directo del dataset que se quiere ingestar:  --dataset-path {sourceFolder}/dataset-{sourceFolder}-{datasetName}/dataset-{sourceFolder}-{datasetName}.json |
| **--source-path** | - | No | Path directo al source, debe seguir la siguiente nomenclatura:  --source-path {sourceFolder}/source-{sourceName}.json  SourceName es el nombre del source definido dentro del metadato del dataset. |
| **--ingest-source (DEPRECADO)** | - | No | **DEPRECADO**  Nombre del origen que se quiere procesar en bloque. Se leeran todos sus dataset y se ingestarán uno a uno. |
| **--config-file** | - | Sí | Ruta desde donde se leerá el fichero de configuración. Con la implementación de UC, debe estar en un volumen del catálogo asociado al entorno:  --config-file /Volumes/{catalogName}/indation/indation/indation-config.yml |
| **--validate-json** | - | No | Ruta padre de todos los ficheros de metadatos (source y dataset) objetivo de la validación. Se usará para validar repositorios de metadatos en integración continua. |

# Motor de metadatos Gold - Configuración.

## Procesamiento de datos

Como norma general, para el procesamiento específico de datos a la capa Gold se levantan casos de uso (proyectos de desarrollo) que se encargan de desarrollar procesos Spark a medida que se alojan en Databricks, de modo que se consiga el procesamiento deseado con el mejor rendimiento posible.

Estos casos de uso requieren el dimensionamiento de un equipo de expertos en el desarrollo de procesos Spark, lo cual no siempre es operativo debido al sobrecoste o a la imposibilidad de aprovisionar un equipo.

Es por este motivo por el que la plataforma dispone de un motor de procesamiento de metadatos para Gold, de modo que usuarios no expertos en procesos Spark puedan configurar procesos de transformación y carga de datos en capa Gold mediante ficheros Json de metadatos similares a los que se utilizan en el motor de ingesta en Silver.

Veremos en este documento cómo se pueden configurar, generar y orquestar estos procesos de transformación con el motor de metadatos gold.

## Metadata

Los metadatos de configuración de procesos de transformación y carga del motor de metadatos gold se organizan en tres tipos de ficheros Json diferentes:

* Conexiones: Definen conexiones con bases de datos Azure SQL de destino final de las transformaciones.
* Actividades: Definen el bloque de orquestación de transformaciones a ejecutar para una actividad específica. El nombre de una actividad debe empezar por “activity-“ y contener el nombre de la carpeta source a continuación: {sourceFolder}/activity-{sourceFolder}-{activityName}/activity-{sourceFolder}-{activityName}.json
* Transformaciones: Definen un bloque completo de transformación de datos basado en etapas de transformación. El nombre de la transformación debe empezar por "transform-" seguido del nombre que se define dentro de la actividad. Se deben guardar en una carpeta con el mismo nombre que la carpeta de la actividad. {sourceFolder}/transform-{transformationName}.json

### Conexiones

Los ficheros de conexión representan todas las posibles bases de datos de Azure SQL a las que puede ir destinada una carga. Hay que tener en cuenta que no todas las transformaciones tienen por qué tener como destino una base de datos Azure SQL, sino que pueden ir orientadas únicamente al almacenamiento de datos preparados en el nivel Datalake.

Patrón de Json

Todos los ficheros de conexiones definidos deben seguir el patrón “sqlconnection\*.json”. Es decir, deben ser de tipo json y depen empezar por el texto “sqlconnection” en minúscula.

Definición de Json

Los campos que se pueden definir en cada fichero de conexión son los siguientes:

|  |  |  |  |
| --- | --- | --- | --- |
| **Campo** | **Tipo** | **Obligatorio** | **Descripción** |
| **name** | String | Sí | Nombre de la conexión. Debe ser único. |
| **description** | String | Sí | Descripción de la conexión. Puede quedarse en blanco, pero debe definirse en el json. |
| **host** | String | Sí | Servidor de Azure SQL donde está la bbdd de destino |
| **port** | String | Sí | Puerto de conexión al servidor |
| **database** | String | Sí | Nombre de la base de datos de destino |

Ejemplo:

{  
 "name": "testConnection",  
 "description":"descripción de testConnection",  
 "typ": {"value": "jdbc"},  
 "host": "xxx-sql-dev-weu-01.database.windows.net",  
 "port": "1433",  
 "database": "xxx-sdb-gld-kpi-operaciones"  
}

## Transformaciones

Los ficheros json de definición de las transformaciones son el grueso de la configuración de transformaciones en Gold mediante el motor de metadatos. Cada json de transformación define un procesamiento por etapas de datos que finaliza en la escritura del set de datos resultante en un destino específico, pudiendo ser éste una tabla de SQL o una ruta en el datalake.

Patrón de Json

Todos los ficheros de conexiones definidos deben seguir el patrón “transform\*.json”. Es decir, deben ser de tipo json y depen empezar por el texto “transform” en minúscula.

### Definición de transformaciones

Los campos que se definen en los json de transformación son los siguientes.

|  |  |  |  |
| --- | --- | --- | --- |
| **Campo** | **Tipo** | **Obligatorio** | **Descripción** |
| **name** | String | Sí | Nombre de la transformación. Debe ser único, no puede definirse otra transformación en la ruta de metadatos con el mismo nombre. |
| **description** | String | Sí | Descripción de la transformación. Puede quedarse en blanco, pero debe definirse en el json. |
| **typ** | TransformationType | Sí | Define el tipo de transforamción mediante un valor de tipo string. Los valores posibles son: - datalake - sql |
| **datalakeTransformation** | Datalake Transformation | No | Bloque de definición de las etapas de transformación con destino en una ruta del datalake. Se requiere cuando el tipo de transformación es "datalake". |
| **sqlTransformation** | SQL Transformation | No | Bloque de definición de las etapas de transformación con destino en SQL. Se requiere cuando el tipo de transformación es "sql". |

Ejemplo:

{  
 "name": "category-dlk",  
 "description":"Bring product category from silver. Simple full copy",  
 "typ": {"value": "datalake"},  
 "datalakeTransformation":  
 {...}  
}

#### Bloque “datalakeTransformation”

En caso de transformaciones de tipo “datalake” se debe definir el bloque específico que configura la transformación con un destino en el datalake y origen de los datos también en el datalake. En el caso de estas transformaciones, el origen puede ser tanto la capa Silver, como la capa Gold en el nivel datalake.

Los campos que se definen en este bloque son los siguientes:

|  |  |  |  |
| --- | --- | --- | --- |
| **Campo** | **Tipo** | **Obligatorio** | **Descripción** |
| **classification** | String | Sí | Nivel de clasificación en que se ubicarán los datos. En MVP es siempre "public" |
| **database** | String | Sí | Nombre de la base de datos de destino (carpeta del datalake) |
| **table** | String | Sí | Nombre de la tabla de destino (carpeta del datalake) |
| **partition** | String | No | Columnas de partición separadas por "/". Tanto si se omite, como si se define con valor en blanco, significa que no se definen particiones para el destino. |
| **mode** | WriteMode | Sí | Modo en que se va a generar la carpeta de destino. Este tipo se usa para todas las escrituras, los valores posibles son: - "append": indica que los valores se van a añadir. - "overwrite": Los valores se van a borrar y cargar de nuevo. En SQL indica que se hace drop de la tabla y se vuelve a crear - "truncate": En SQL indica que la tabla se trunca antes de cargarse, no se elimina. |
| **stages** | List[DatalakeTransformationStage] | Sí | Lista con todas las etapas de transformación que se realizan antes de escribir el destino |
| **qualityRules** | QualityRules | No | Define las reglas de calidad a aplicar en la capa gold (ver descripción en la sección de silver “QualityRules” |

La lista de etapas especifica una lista ordenada en la que cada etapa que se ejecuta realiza una operación nueva en el dataframe resultante de la etapa anterior, de modo que se va moldeando un juego de datos cada vez más refinado que, cuando llega al último paso, escribe todo el resultado en el destino espeficiado por los campos de clasificación/database/table.

Cada una de las etapas que se definen pueden tener los siguientes campos:

|  |  |  |  |
| --- | --- | --- | --- |
| **Campo** | **Tipo** | **Obligatorio** | **Descripción** |
| **name** | String | Sí | Nombre de la etapa de transformación. Es meramente descriptivo. |
| **description** | String | Sí | Descripción de la etapa de transformación |
| **typ** | StageType | Sí | Tipo de etapa de transformación. Los tipos posibles son los siguientes: - table: traer una nueva tabla o expresión sql al set de datos - calculatedColumn: genera una nueva columna en el set de datos a partir de una expresión - select: selecciona una lista de columnas - aggregation: realiza una agregación sobre el set de datos anterior - filter: aplica un filtrado sobre el set de datos |
| **tempView** | String | No | Indica un nombre de una vista que se va a crear con el resultado de la etapa. Se recomienda ser riguroso con estos nombres de vistas temporales, ya que si se usa algo genérico se pueden producir problemas de sincronización con varias ejecuciones paralelas. |
| **distinct** | Boolean | No | Si se define con el valor **true**, al set de datos resultante se le aplica un distinct. |
| **stageTable** | StageTable | No | Definición de la etapa de tipo "table". |
| **stageColumn** | StageColumn | No | Definición de la etapa de tipo "calculatedColumn" |
| **stageSelect** | List[StageSelect] | No | Definición de la etapa de tipo "select" |
| **stageFilter** | StageFilter | No | Definición de la etapa de tipo "filter" |
| **stageAggregation** | StageAggregation | No | Definición de la etapa de tipo "aggregation" |
| **drop** | List[String] | No | Lista de columnas de eliminar |
| **stageFiller** | StageFiller | No | Definición de la etapa de tipo “StageFiller” |
| **stageBinarizer** | StageBinarizer | No | Definición de la etapa de tipo “StageBinarizer” |
| **stageBucketizer** | StageBucketizer | No | Definición de la etapa de tipo “StageBucketizer” |
| **stageVectorIndexer** | StageVectorIndexer | No | Definición de la etapa de tipo “StageVectorIndexer” |
| **stageOneHotEncoder** | StageOneHotEncoder | No | Definición de la etapa de tipo “StageOneHotEncoder” |
| **stageImputer** | StageImputer | No | Definición de la etapa de tipo “StageImputer” |
| **stageVariation** | StageVariation | No | Definición de la etapa de tipo “StageVariation” |
| **stageMonthlyConverter** | StageMonthlyConverter | No | Definición de la etapa de tipo “StageMonthlyConverter” |
| **stageLambda** | StageLambda | No | Definición de la etapa de tipo “StageLambda” |

Como se comenta en la tabla, cada tipo diferente de etapa tiene su propia definición particular. Cada una de estas definiciones son las que realizan el tratamiento específico sobre el set de datos de la etapa anterior.

Pasamos a detallar cada una de estas posibles definiciones de etapas:

##### “stageTable”

Define una etapa que trae un nuevo bloque de datos al set de datos, ya sea importando una tabla, ya sea a través de la ejecución de una sentencia SQL de Spark.

|  |  |  |  |
| --- | --- | --- | --- |
| **Campo** | **Tipo** | **Obligatorio** | **Descripción** |
| **name** | String | Sí | Nombre de la etapa |
| **typ** | TableType | Sí | Tipo de tabla. Los valores posibles son: - "silver": para indicar que es una tabla de la capa Silver - "gold": para indicar que es una tabla generada en capa Gold - "sql": para indicar que es una expresión sql de spark |
| **database** | String | No | Para los tipos "silver" o "gold", especifica el nombre de la base de datos donde se aloja la tabla a importar. |
| **table** | String | Sí | Indica el nombre de la tabla a importar. El dataset de salida tiene este nombre como alias. En caso de una expresión SQL no especifica realmente un origen, pero sí tiene importancia desde el punto de vista de generación del alias. |
| **classification** | String | No | En caso de tipo de tabla "gold" se debe especifica la clasificación para poder componer la ruta completa. En MVP será siempre "public" |
| **sql** | String | No | En caso de tipo de expresión "sql" se define la expresión en este campo. |
| **joinExpr** | JoinExpr | No | Cuando el set de datos se debe combinar con un set de datos resultante de la etapa anterior, se debe definir esta combinación como una expresión de join. Si no se define la expresión de join, el resultado del set de datos sobrescribe cualquier set de la etapa anterior, por lo que lo normal es que sólo quede sin definir en la primera etapa de una transformación. |

La expresión join se define mediante el tipo de join y la expresión de la condición de join

|  |  |  |  |
| --- | --- | --- | --- |
| **Campo** | **Tipo** | **Obligatorio** | **Descripción** |
| **typ** | JoinType | Sí | Tipo de join: - inner - cross - fullouter - leftanti - leftouter - rightouter - leftsemi |
| **expr** | String | Sí | Expresión spark sql que define la condición específica de join |

##### “stageColumn”

La etapa de tipo columna calculada se define a partir de la expresión sql que genera el resultado del cálculo de la nueva columna a generar.

|  |  |  |  |
| --- | --- | --- | --- |
| **Campo** | **Tipo** | **Obligatorio** | **Descripción** |
| **name** | String | Sí | Nombre de la columna calculada |
| **description** | String | Sí | Descripción del cálculo |
| **typ** | CalculationType | Sí | Tipo de cálculo. Actualmente sólo es posible el tipo "expr" para definir una expresión spark sql |
| **calculationExpr** | String | Sí | Expresión spark sql que define el cálculo |

##### “stageSelect”

La etapa de tipo select se define con una lista de columnas que se quieren seleccionar del set de datos resultante de la etapa anterior. Suele ser habitual en la última etapa, donde ya se quieren discriminar las columnas a escribir en el destino y definir sus nombres finales.

|  |  |  |  |
| --- | --- | --- | --- |
| **Campo** | **Tipo** | **Obligatorio** | **Descripción** |
| **columnName** | String | Sí | Nombre de la columna |
| **columnDescription** | String | Sí | Descripción de la columna |
| **columnAlias** | String | Sí | Alias a aplicar a la columna |

##### stageFilter

Define un filtro que se aplica sobre el resultado de la etapa anterior. Es el equivalente a un “where” en cualquier consulta sql.

|  |  |  |  |
| --- | --- | --- | --- |
| **Campo** | **Tipo** | **Obligatorio** | **Descripción** |
| **typ** | FilterType | Sí | Tipo de filtro. Actualmente sólo pueden ser de tipo "expr": expresión sql |
| **filterExpr** | String | Sí | String con la expresión SQL que especifica la condición a aplicar |

##### stageAggregation

Las agregaciones se definen mediante el set de columnas que se agrupan y las columnas de agregación.

|  |  |  |  |
| --- | --- | --- | --- |
| **Campo** | **Tipo** | **Obligatorio** | **Descripción** |
| **name** | String | Sí | Nombre de la etapa de agregación |
| **description** | String | Sí | Descripción de la etapa |
| **stageGroupBy** | List[String] | Sí | Lista de nombres de columnas que forman la agrupacion. Equivale a un "group by" |
| **aggregations** | List[AggregationColumn] | Sí | Lista de columnas de agregación a calcular en base a la agrupación definida. |

La lista de columnas de agregación se define a partir del tipo de agregación y la columna utilizada para la agregación

|  |  |  |  |
| --- | --- | --- | --- |
| **Campo** | **Tipo** | **Obligatorio** | **Descripción** |
| **name** | String | Sí | Nombre de la columna a rellenar valores vacíos |
| **description** | String | Sí | Descripción de la columna |
| **typ** | AggregationType | Sí | Tipo de agregación: - sum - sumDistinct - count - countDistinct - min - max - first - last - avg - mean |
| **col** | String | Sí | Nombre de la columna usada en el cálculo de la agregación. |

##### StageFiller

La etapa de tipo StageFiller define el valor a rellenar en los campos vacíos para las columnas seleccionadas.

|  |  |  |  |
| --- | --- | --- | --- |
| **ampo** | **Tipo** | **Obligatorio** | **Descripción** |
| **name** | String | Sí | Nombre de la transformación StageFiller |
| **description** | String | Sí | Descripción de la transformación |
| **typ** | FillerInputTypes | Sí | Tipo de dato de la columna a rellenar:   * String * Double * Int * Boolean * Long |
| **value** | String |  | Valor utilizado para rellenar los valores vacíos |
| **cols** | Array[String] | No | Columnas sobre las que realizar el relleno de campos vacíos |

##### StageBinarizer

La etapa de tipo StageBinarizer define la binarización de columnas de tipo continuo seleccionadas dado un umbral especificado.

|  |  |  |  |
| --- | --- | --- | --- |
| **Campo** | **Tipo** | **Obligatorio** | **Descripción** |
| **name** | String | Sí | Nombre de la transformación StageBinarizer. |
| **description** | String | Sí | Descripción de la transformación. |
| **typ** | ColsTypes | Sí | Indica si es una columna a binarizar o múltiples:   * single * list |
| **threshold** | Double | No | Umbral a definir. Su valor por defecto es 0 |
| **thresholds** | Array[Double] | No | Umbrales |
| **inputCol** | String | No | Columna de entrada |
| **inputCols** | Array[String] | No | Columnas de entrada |
| **outputCol** | String | No | Columna de salida |
| **outputCols** | Array[String] | No | Columnas de salida |

##### StageBucketizer

La etapa de tipo StageBucketizer define las columnas de tipo continuo para ser mapeadas a columnas de tipo bucket.

|  |  |  |  |
| --- | --- | --- | --- |
| **Campo** | **Tipo** | **Obligatorio** | **Descripción** |
| **name** | String | Sí | Nombre de la transformación StageBucketizer. |
| **description** | String | Sí | Descripción de la transformación. |
| **typ** | ColsTypes | Sí | Indica si es una columna a bucketizar o múltiples:   * single * list |
| **splits** | Array[Double] | No | Divisiones a definir. |
| **splitsArrrays** | Arrray[Array[Double]] | No | Arrays de divisiones a definir |
| **inputCol** | String | No | Columna de entrada |
| **inputCols** | Array[String] | No | Columnas de entrada |
| **outputCol** | String | No | Columna de salida |
| **outputCols** | Array[String] | No | Columnas de salida |

##### StageVectorIndexer

La etapa de tipo StageVectorIndexer define la columna de tipo categórico a indexar en un conjunto de datos de Vector.

|  |  |  |  |
| --- | --- | --- | --- |
| **Campo** | **Tipo** | **Obligatorio** | **Descripción** |
| **name** | String | Sí | Nombre de la transformación StageVectorIndexer. |
| **description** | String | Sí | Descripción de la transformación. |
| **historicalTable** | StageTable | Sí | Tabla histórica de tipo StageTable (ver descripción de stageTable más arriba en el documento) |
| **inputCol** | String | No | Columna de entrada |
| **outputCol** | String | No | Columna de salida |
| **maxCategories** | Int | No | Número máximo de categorías. Valor por defecto de 20 |

##### StageOneHotEncoder

La etapa de tipo StageOneHotEncoder define una columna categórica a mapear como columna de vectores binarios.

|  |  |  |  |
| --- | --- | --- | --- |
| **Campo** | **Tipo** | **Obligatorio** | **Descripción** |
| **name** | String | Sí | Nombre de la transformación StageOneHotEncoder. |
| **description** | String | Sí | Descripción de la transformación. |
| **typ** | ColsTypes | Sí | Indica si es una columna sobre la que se realiza el enconding o múltiples:   * single * list |
| **historicalTable** | StageTable | Sí | Tabla histórica de tipo StageTable (ver descripción de stageTable más arriba en el documento) |
| **inputCol** | String | No | Columna de entrada |
| **inputCols** | Array[String] | No | Columnas de entrada |
| **outputCol** | String | No | Columna de salida |
| **outputCols** | Array[String] | No | Columnas de salida |

##### StageImputer

La etapa de tipo StageImputer define las columnas sobre las cuales se imputarán los valores vacíos, así como la técnica a aplicar.

|  |  |  |  |
| --- | --- | --- | --- |
| **Campo** | **Tipo** | **Obligatorio** | **Descripción** |
| **name** | String | Sí | Nombre de la transformación StageImputer. |
| **description** | String | Sí | Descripción de la transformación. |
| **typ** | ColsTypes | Sí | Indica si es una columna a imputar o múltiples:   * single * list |
| **strategy** | ImputerTypes | No | Tipo de imputación:   * Mean (defecto) * Median * Mode |
| **historicalTable** | StageTable | No | Tabla histórica de tipo StageTable (ver descripción de stageTable más arriba en el documento) |
| **inputCol** | String | No | Columna de entrada |
| **inputCols** | Array[String] | No | Columnas de entrada |
| **outputCol** | String | No | Columna de salida |
| **outputCols** | Array[String] | No | Columnas de salida |

##### StageVariation

La etapa StageVariation define un lag de tiempo que se aplicará a la columna de entrada en base a la columna timeOrderCol. La variación quedará recogida en la columna de salida.

|  |  |  |  |
| --- | --- | --- | --- |
| **Campo** | **Tipo** | **Obligatorio** | **Descripción** |
| **name** | String | Sí | Nombre de la transformación StageVariation. |
| **description** | String | Sí | Descripción de la transformación. |
| **percentage** | Boolean | Sí | Si es true, se calcula la variación de manera porcentual. False por defecto |
| **timeLag** | Int | Sí | Lag de tiempo a aplicar |
| **timeOrderCol** | String | Sí | Columna en la que la columna de entrada aplicará el lag |
| **inputCol** | String | Sí | Columna de entrada |
| **outputCol** | String | Sí | Columna sobre la que se aplicará el resultado |

##### StageMonthlyConverter

|  |  |  |  |
| --- | --- | --- | --- |
| **Campo** | **Tipo** | **Obligatorio** | **Descripción** |
| **name** | String | Sí | Nombre de la transformación StageMonthlyConverter. |
| **description** | String | Sí | Descripción de la transformación. |
| **freq** | MonthlyTypes | Sí | Frequencia:   * Bi-monthly * Quarterly * Four-monthly * Six-monthly |
| **dateCol** | String | Sí | Columna de tipo fecha |
| **idCol** | String | Sí | Columna sobre la que se aplicará particionado |

##### Ejemplo Datalake Transformation

A continuación, un ejemplo completo de una transformación de datalake con todos los tipos de etapa involucrados.

{  
 "name": "product-subcategory-dlk",  
 "description":"product joined with subcategory, both from silver",  
 "typ": {"value": "datalake"},  
 "datalakeTransformation":  
 {  
 "classification": "public",  
 "database": "production",  
 "table": "product-subcategory",  
 "partition": "ProductName/sum",  
 "mode": {"value": "overwrite"},  
 "stages": [  
 {  
 "name": "stage table 1",  
 "description": "stage 1",  
 "typ": {"value": "table"},  
 "stageTable": {  
 "name": "DimProduct",  
 "typ": {"value": "silver"},  
 "database": "database",  
 "table": "DimProduct"  
 }  
 },  
 {  
 "name": "stage table 1 spark sql",  
 "description": "stage 1 bis, hace lo mismo que la anterior, pero vía spark.sql",  
 "typ": {"value": "table"},  
 "stageTable": {  
 "name": "DimProduct",  
 "typ": {"value": "sql"},  
 "table": "DimProduct",  
 "sql": "select \* from database.DimProduct"  
 }  
 },  
 {  
 "name": "stage table 2",  
 "description": "stage 1.2",  
 "typ": {"value": "table"},  
 "stageTable": {  
 "name": "DimProductSubcategory",  
 "typ": {"value": "gold"},  
 "database": "database",  
 "table": "DimProductSubcategory",  
 "classification": "public",  
 "joinExpr": {  
 "typ": {"value": "inner"},  
 "expr": "DimProductSubcategory.ProductSubcategoryKey = DimProduct.ProductSubcategoryKey"  
 }  
 }  
 },  
 {  
 "name": "stage 2 calculatedColumn",  
 "description": "stage 2",  
 "typ": {"value": "calculatedColumn"},  
 "stageColumn": {  
 "name": "ProductName",  
 "description": "Product Description",  
 "typ": {"value": "expr"},  
 "calculationExpr": "DimProduct.EnglishProductName"  
 }  
 },  
 {  
 "name": "stage select",  
 "description": "stage 3",  
 "typ": {"value": "select"},  
 "stageSelect": [  
 {  
 "columnName": "DimProduct.ProductKey",  
 "columnDescription": "ProductKey",  
 "columnAlias": "ProductKey"  
 },  
 {  
 "columnName": "DimProductSubcategory.ProductSubcategoryKey",  
 "columnDescription": "ProductSubcategoryKey",  
 "columnAlias": "ProductSubcategoryKey"  
 },  
 {  
 "columnName": "ProductName",  
 "columnDescription": "ProductName",  
 "columnAlias": "ProductName"  
 }  
 ]  
 },  
 {  
 "name": "stage aggregation",  
 "description": "stage 4",  
 "typ": {"value": "aggregation"},  
 "stageAggregation": {  
 "name": "agg 1",  
 "description": "agg 1",  
 "stageGroupBy": ["ProductName"],  
 "aggregations": [  
 {  
 "name": "sum",  
 "description": "agg",  
 "typ": {"value": "sum"},  
 "col": "ProductKey"  
 },  
 {  
 "name": "sumDistinct",  
 "description": "agg",  
 "typ": {"value": "sumDistinct"},  
 "col": "ProductKey"  
 },  
 {  
 "name": "count",  
 "description": "agg",  
 "typ": {"value": "count"},  
 "col": "ProductKey"  
 }  
 ]  
 }  
 },  
 {  
 "name": "stage filter HL Road Frame - Black, 58",  
 "description": "stage 5",  
 "typ": {"value": "filter"},  
 "tempView": "vw\_result",  
 "stageFilter": {  
 "typ": {"value": "expr"},  
 "filterExpr": "ProductName = 'HL Road Frame - Black, 58'"  
 }  
 },  
 {  
 "name": "stage final select spark sql",  
 "description": "stage 6",  
 "typ": {"value": "table"},  
 "distinct": true,  
 "stageTable": {  
 "name": "FinalTable",  
 "typ": {"value": "sql"},  
 "table": "FinalTableName",  
 "sql": "select ProductName, sum, sumDistinct, count from vw\_result"  
 }  
 }  
 ]  
 }  
}

### Bloque “sqlTransformation”

En caso de transformaciones de tipo “sql” se debe definir el bloque específico que configura la transformación con un destino en Azure SQL. Este tipo de transformación se define a partir de una lista de tablas importadas en bruto desde el datalake y una lista de etapas de transformación basadas en expresiones t-sql.

|  |  |  |  |
| --- | --- | --- | --- |
| **Campo** | **Tipo** | **Obligatorio** | **Descripción** |
| **sourceTables** | List[GoldTableSource] | Sí | Lista de tablas del datalake a importar. |
| **stages** | List[SQLTransformationStage] | Sí | Lista de etapas de transformación a ejecutar. |

#### GoldTableSource

La lista de importación de tablas desde el datalake genera tablas en SQL con la información especificada. Estas tablas pueden tratarse como tablas staging o incluso como tablas definitivas, ya que en muchos casos se puede desear únicamente el copiado de tablas del datalake tal y como están.

Se recomienda, tanto en estas tablas, como en las de destino de etapas de transformación, que las tablas hayan sido previamente definidas y creadas en la base de datos y que no se utilice el modo “overwrite” si está previsto utilizar las tablas en explotación. Esto permitirá ser más riguroso con los tipos de datos, la indexación y la optimización relacional en general.

|  |  |  |  |
| --- | --- | --- | --- |
| **Campo** | **Tipo** | **Obligatorio** | **Descripción** |
| **dlkClassification** | String | No | En caso de ser un origen de tipo tabla gold del datalake, la ruta de clasificación. En MVP es siempre "public". |
| **dlkDatabase** | String | No | En caso de tabla gold datalake se define la ruta de nombre de base de datos. |
| **dlkTable** | String | No | En caso de tabla gold datalake se define la ruta de nombre de tabla de origen. |
| **dlkSQL** | String | No | En caso de querer importar la información a partir de una expresión spark SQL, se define este campo con la expresión de consulta. Esta expresión no usa parámetros de actividad. |
| **dlkFilterExpr** | String | No | Expresión de filtrado del resultado de la tabla o consulta spark sql importada. Esta expresión sí admite uso de parámetros de actividad. |
| **sqlConnection** | String | Sí | Nombre de la conexión de destino de la base de datos en la que se va a importar el contenido del datalake. |
| **sqlSchema** | String | Sí | Nombre del esquema de destino. |
| **sqlTable** | String | Sí | Nombre de la tabla de destino. |
| **sqlWriteMode** | WriteMode | Sí | Modo de escritura de la tabla de destino.  - "append": indica que los valores se van a añadir. - "overwrite": Los valores se van a borrar y cargar de nuevo. En SQL indica que se hace drop de la tabla y se vuelve a crear - "truncate": En SQL indica que la tabla se trunca antes de cargarse, no se elimina. |
| **cleanAfter** | Boolean | Sí | Si se define a true, al finalizar la transformación se hará truncado de esta tabla. |
| **dropAfter** | Boolean | Sí | Si se define a true, al finalizar la transformación se hará drop de esta tabla. |

#### SQLTransformationStage

La lista de etapas de transformación SQL implica una lista que se ejecuta de manera ordenada, pero que, a diferencia de las transformaciones del datalake, no aplica cada transformación al resultado de la ejecución anterior, sino que siempre define una tabla de destino y una expresión SQL a ejecutar en el motor de base de datos.

|  |  |  |  |
| --- | --- | --- | --- |
| **Campo** | **Tipo** | **Obligatorio** | **Descripción** |
| **connection** | String | Sí | Nombre de la conexión a utilizar como bbdd SQL de destino. Debe coincidir con el nombre de una conexión definida en metadata. |
| **destSchema** | String | Sí | Esquema de destino en la bbdd. Es siempre obligatorio. |
| **destTable** | String | Sí | Nombre de la tabla de destino en bbdd. |
| **sqlSelectExpr** | String | Sí | Expresión T-SQL a ejecutar en el motor de base de datos para extraer la información. |
| **filterExpr** | String | No | Expresión de condición de filtrado sobre el resultado de la select anterior. Aunque la expresión de select puede contener la cláusula where, el filtrado posterior en modo spark permite la aplicación de parámetros de actividad (ver más adelante). |
| **writeMode** | WriteMode | Sí | Modo de escritura ya descrito anteriormente:  - "append": indica que los valores se van a añadir. - "overwrite": Los valores se van a borrar y cargar de nuevo. En SQL indica que se hace drop de la tabla y se vuelve a crear - "truncate": En SQL indica que la tabla se trunca antes de cargarse, no se elimina. |
| **cleanAfter** | Boolean | Sí | Si está definido con valor **true**, al finalizar la transformación se trunca esta tabla. |
| **dropAfter** | Boolean | Sí | Si está definido con valor **true**, al finalizar la transformación se elimina esta tabla. |

#### Ejemplo de transformación SQL

A continuación, se muestra un ejemplo representativo de transformación SQL.

{  
 "name": "plain-product-sql",  
 "description":"product joined with category and subcategory",  
 "typ": {"value": "sql"},  
 "sqlTransformation":  
 {  
 "sourceTables": [  
 {  
 "dlkClassification": "public",  
 "dlkDatabase": "production",  
 "dlkTable": "category",  
 "dlkFilterExpr": "ProductCategoryKey <= 3",  
 "sqlConnection": "testConnection",  
 "sqlSchema": "staging",  
 "sqlTable": "category",  
 "sqlWriteMode": {"value": "append"},  
 "cleanAfter": false,  
 "dropAfter": true  
 },  
 {  
 "dlkSQL": "select ProductName, sum, sumDistinct, count from vw\_result",  
 "sqlConnection": "testConnection",  
 "sqlSchema": "staging",  
 "sqlTable": "product\_subcategory",  
 "sqlWriteMode": {"value": "overwrite"},  
 "cleanAfter": true,  
 "dropAfter": false  
 }  
 ],  
 "stages": [  
 {  
 "connection": "testConnection",  
 "destSchema": "datamart",  
 "destTable": "plainProduct",  
 "sqlSelectExpr": "select \* from staging.product\_subcategory",  
 "writeMode": {"value": "truncate"},  
 "cleanAfter": true,  
 "dropAfter": false  
 },  
 {  
 "connection": "testConnection",  
 "destSchema": "datamart",  
 "destTable": "category",  
 "sqlSelectExpr": "select \* from staging.category",  
 "filterExpr": "paramCategory = ProductCategoryKey",  
 "writeMode": {"value": "overwrite"},  
 "cleanAfter": false,  
 "dropAfter": true  
 },  
 {  
 "connection": "testConnection",  
 "destSchema": "datamart",  
 "destTable": "aggProduct",  
 "sqlSelectExpr": "select productName, count(\*) total from datamart.category cross join datamart.plainProduct group by productName",  
 "writeMode": {"value": "overwrite"},  
 "cleanAfter": false,  
 "dropAfter": false  
  
 }  
 ]  
 }  
}

## Actividades

El tercer bloque de archivos json que se deben definir en los metadatos del motor gold es la definición de las actividades de orquestación de transformaciones.

Estas actividades definen la lista ordenada de transformaciones que deben ejecutarse de manera concurrente y la lista de parámetros que se espera recibir cuando se invoca la actividad.

Las actividades serán invocadas directamente desde Data Factory como se verá más adelante.

Patrón de Json

Todos los ficheros de actividades definidos deben seguir el patrón “activity\*.json”. Es decir, deben ser de tipo json y depen empezar por el texto “activity” en minúscula.

Definición de Json

El json de actividad se define con un nombre identificativo único, la lista de transformaciones y la lista de parámetros.

|  |  |  |  |
| --- | --- | --- | --- |
| **Campo** | **Tipo** | **Obligatorio** | **Descripción** |
| **name** | String | Sí | Nombre de la actividad. Este nombre debe ser único, ya que es el que se utiliza para invocar la actividad desde Data Factory. |
| **description** | String | Sí | Descripción de la actividad. |
| **transformations** | List[String] | Sí | Lista ordenada de nombres de transformaciones a ejecutar. |
| **parameters** | List[Parameter] | No | Listado de parámetros que se esperan en la invocación de la actividad. |

### Parámetros de actividad

Los parámetros de actividad definen una lista de valores que deberá ser enviada desde Data Factory cuando se invoca la actividad y que serán trasladados a todas las etapas de transformación que se ejecutan en toda la actividad.

Al ser trasladados estos valores directamente, se proporciona la posibilidad de utilizar estos parámetros como columnas en todos los sets de datos, de modo que sea posible utilizarlos en todas las expresiones que se ejecuten (filtros, columnas calculadas, selects, etc.).

Siempre que se escribe en la tabla de destino el resultado de una transformación se eliminan las columnas de parámetros.

|  |  |  |  |
| --- | --- | --- | --- |
| **Campo** | **Tipo** | **Obligatorio** | **Descripción** |
| **shortName** | String | Sí | Nombre corto del parámetro. Especifica que se invoca con un "-" delante (ej. "-param valor") |
| **longName** | String | Sí | Nombre largo del parámetro. Especifica que se invoca con un "--" delate (ej. "--parametro valor"). Este nombre largo es el que se añade como nombre de columna a los sets de datos para poder ser utilizado en expresiones de las etapas de transformación. |
| **description** | String | Sí | Descripción del parámetro. |

### Ejemplo de actividad

A continuación, se muestra un ejemplo representativo de un json de actividad

{  
 "name": "testActivity",  
 "description":"activity for testing",  
 "transformations": [  
 "product-subcategory-dlk",  
 "category-dlk",  
 "plain-product-sql"  
 ],  
 "parameters": [  
 {  
 "shortName": "paramCat",  
 "longName": "paramCategory",  
 "description": "Param Category"  
 }  
 ]  
}

## Ejecución

### Parámetros de entrada

El motor podrá recibir parámetros de ejecución a través de la línea de comandos. En el caso de ser lanzados a través del orquestador, estos también podrán ser inyectados por el mismo.

La especificación de los parámetros de entrada seguirá la convención GNU. De esta manera los parámetros son auto explicativos y es fácil determinar su función.

|  |  |  |
| --- | --- | --- |
| Parámetro | Obligatorio | Descripción |
| **--adf-runid** | No | Id de ejecución del pipeline de Data Factory. De esta manera podemos vincular la ejecución del motor con la ejecución del pipeline que lo invocó. Se puede obtener de la variable de Data Factory @pipeline().RunId |
| **--transformation-activity** | Sí | Nombre de la actividad que se invoca por el orquestador para ser ejecutada. |
| **--config-file** | Sí | Ruta desde donde se leerá el fichero de configuración. |
| **--publish-statistics** | No | En caso de estar definido como false, las estadísticas no serán publicadas |
| **--validate-transformations** | No | Valida la estructura de los ficheros de transformación |
| **--validate-activities** | No | Valida la estructura de los ficheros de actividad |
| **--validate-connections** | No | Valida la estructura de las conexiones |

Adicionalmente, se deben añadir los parámetros de entrada específicos de la actividad invocada (ver apartado 3.4.1)