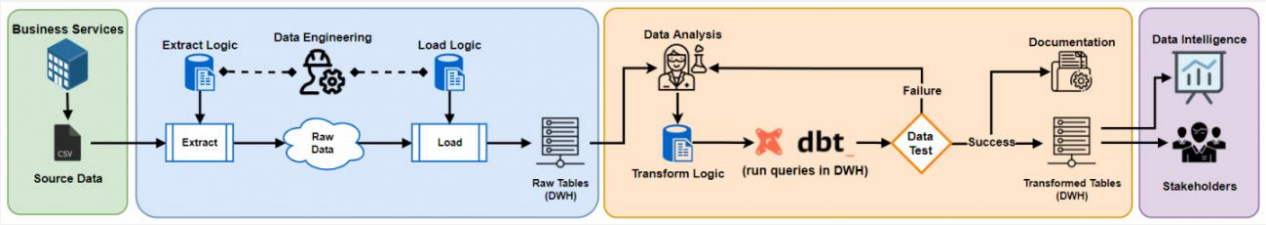
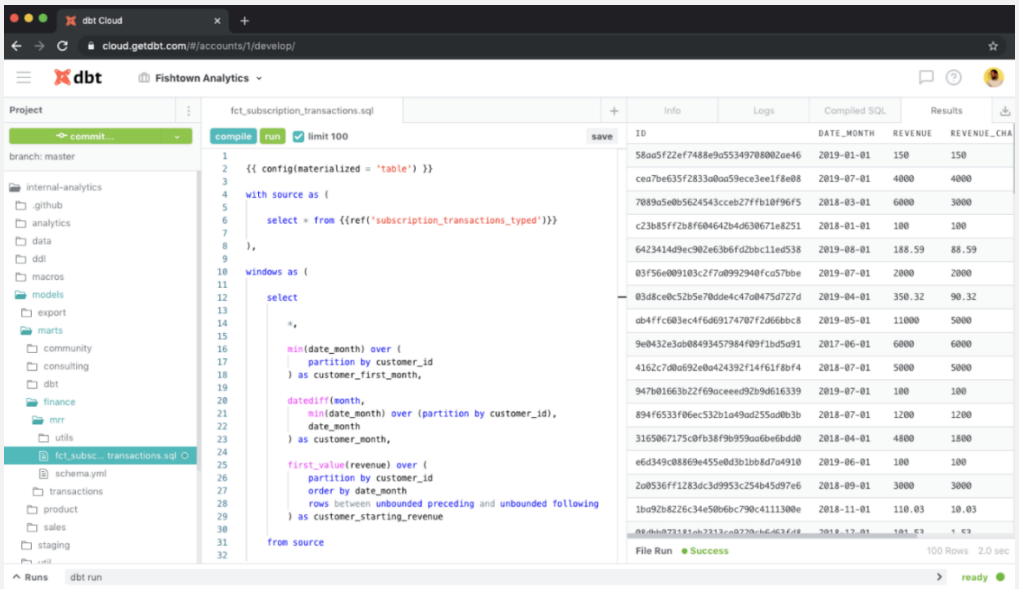
DBT – Cómo funciona





CONTENIDO

1. INTRODUCCIÓN

En la era de la información, el análisis de datos se ha convertido en una parte esencial para cualquier organización, independientemente de su ámbito o propósito. La extracción de datos y su posterior análisis, permiten tomar decisiones informadas y apropiadas para cada situación.

Para soportar el análisis de datos, el primer paso es la extracción, transformación y carga de datos (procesos ETL o ELT para Big Data) desde las fuentes hasta los almacenes de datos (Data Warehouses) destino. En estos, la velocidad, y la interoperabilidad entre herramientas y sistemas de almacenamiento de datos es vital.

Para dar respuesta a esta necesidad, en las últimas décadas han emergido un gran número de herramientas de tipo ETL o ELT. Entre ellas destaca desde hace relativamente poco tiempo **Data Build Tool (DBT)**, herramienta basada en código libre que busca agilizar el proceso de transformación de datos promoviendo la reutilización y el uso de código altamente legible.

Sin embargo, estas herramientas suelen tener que abrirse paso entre herramientas más veteranas que han demostrado su robustez y funcionalidades a lo largo de muchos años. Por este motivo, en el presente estudio, analizamos y ponemos a prueba la herramienta DBT, pero además la comparamos con una herramienta más tradicional y ampliamente usada en la industria, Pentaho Data Integration.

2. ¿QUÉ ES DBT?

Data Build Tool es una herramienta de transformación de datos de código abierto, que se utiliza para el análisis de datos. DBT emplea una combinación de SQL y Jinja para definir transformaciones de datos, que se ejecutan posteriormente sobre la plataforma de datos de la empresa. Jinja es un motor de plantillas (*templating*) para Python que permite escribir código reutilizable mediante etiquetas, que reciben datos a posteriori para su procesamiento.

Además, utiliza YAML para definir los metadatos de las transformaciones. El uso de estos lenguajes tan comunes y cercanos al lenguaje natural facilita su entendimiento y agiliza el proceso de familiarización.

DBT facilita la gestión del ciclo de vida de la transformación de datos, desde la extracción hasta la carga en el sistema de destino, permitiendo a los equipos de desarrollo implementar y mantener pipelines de datos con mayor eficiencia. Esto lo consigue gracias a su integración con GIT y su capacidad para crear conjuntos de pruebas.

DBT también tiene un sistema documentación propio que ayuda a garantizar que el código sea más fácil de mantener y entender. Este sistema permite generar paginas HTML que explican el funcionamiento de las transformaciones de una manera sencilla, permitiendo navegar entre el resto de los elementos documentados a través de una interfaz gráfica.

Además, se complementa con el servicio DBT Hub que provee a los usuarios con paquetes de código. Estos paquetes proveen a los usuarios código genérico y reutilizable. Dichos paquetes cumplen una función similar a las librerías en otros lenguajes de programación.

3. INFORMACION FUNCIONAL Y TÉCNICA

¿Cómo funciona DBT?

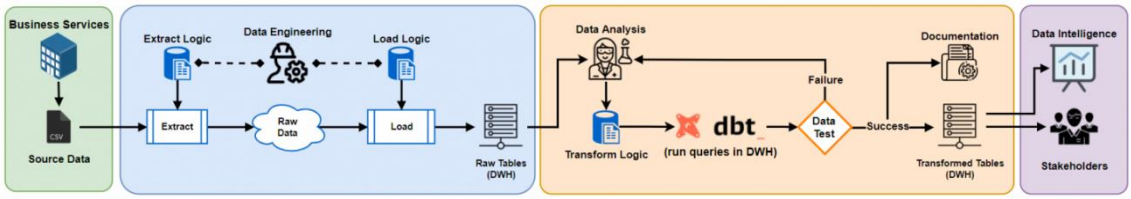


Figura 1: DBT en el proceso ELT

Como se puede apreciar en la Figura 1, DBT es una herramienta que se usa en los procesos conocidos como ETL (Extracción, Transformación y Carga) o ELT (Extracción, Carga y Transformación):

- Se extraen los datos en bruto de una o varias fuentes que pueden variar en cuanto a su naturaleza (Por ejemplo: bases de datos y ficheros Excel).
- Se organizan, se eliminan los datos irrelevantes y se transforman los datos restantes en una serie de conjuntos de datos analizables.
- Finalmente se cargan en el destino final para su posterior análisis.

DBT puede manejar orígenes de datos relacionales como PostgreSQL o SQL Server. Las fuentes SQL se definen junto a los modelos en un fichero llamado “Schema.yml”.

También se pueden añadir datos utilizando *Seeds (Semillas), estas son* pequeñas colecciones de datos poco variables en el tiempo (estáticas), que se añaden al proyecto utilizando un fichero CSV. Cuando se ejecuta un modelo en DBT que utiliza una semilla, DBT primero carga los datos de la semilla en una tabla de la base de datos, y luego utiliza los datos de esa tabla como entrada para el modelo. Un ejemplo de Seed podría ser una categorización de clientes en base a algunos de sus atributos (ej. volumen de compras por año).

DBT se puede ejecutar de dos maneras: mediante su plataforma DBT Cloud, o de manera local usando DBT Core. Una las características principales de DBT, es que el procesamiento se ejecuta sobre el motor de la base de datos o sistema seleccionado. Entre los disponibles, encontramos principalmente sistemas de procesamiento con capacidades Big Data como Databricks/Apache Spark, Snowflake, BigQuery, Redshift o Azure Synapse, aunque también se soporta la BD PostgreSQL.

Una vez hemos creado un proyecto DBT Core, podemos ejecutarlo usando la línea de comandos. No obstante, DBT es compatible con orquestadores como Airflow que facilitan la ejecución planificada de múltiples procesos con posibles dependencias.

Otra de las características de DBT, es el soporte para extensiones en formato de platillas Jinja para Python, pudiendo descargar extensiones ya existentes o crear la nuestras propias con Python.

Por otro lado, es importante resaltar que en la versión open-source se requiere de un IDE o un editor de texto para trabajar, mientras que DBT Cloud ofrece un entorno de desarrollo web propio.

¿Cuáles son las principales características de DBT?

1) **Centrado en la transformación**: se integra perfectamente con herramientas de extracción tales como Fivetran. De esta manera, los usuarios pueden centrarse exclusivamente en crear modelos de transformación correctamente estructurados y adecuados.

2) **Basado en Código**: mediante el uso de lenguajes comunes y sencillos como SQL y Jinja/Python, se puede crear código reutilizable, comprensible, escalable y comprobable.

3) **Administración del ciclo de vida del proyecto**: en DBT se puede configurar un pipeline CI/CD (Integración Continua / Despliegue Continuo) gracias a su fácil integración con sistemas de control de versiones como GIT, su planificador (en la versión Cloud), y a su capacidad de crear pruebas específicas y una suite propia para realizar pruebas unitarias.

4) **Código reutilizable con Jinja**: DBT permite crear macros usando Jinja/Python. Las macros son fragmentos de código pensados para ser reutilizados. Son básicamente scripts a los que se puede llamar en múltiples partes del código, como, por ejemplo, en las pruebas. El paquete "dbt_utils" ofrece no solo una serie de transformaciones para realizar pruebas unitarias, sino que además proporciona macros y generadores SQL que nos ahorrarán tiempo durante el desarrollo.

DBT Core vs DBT Cloud:

Como comentamos anteriormente, existen dos versiones de DBT: Core y Cloud. La principal diferencia entre ambas versiones es que Core se instala de manera local y se trabaja a través de editores de código tales como Microsoft Visual Studio Code o Sublime. Por otro lado, la versión Cloud ofrece un entrono web propio al que se puede acceder utilizando cualquier navegador. En ambas versiones, el desarrollo se hace mediante código.

DBT Cloud tiene dos modalidades de uso:

- **Multi-tenant (Múltiples-inquilinos):** Se utiliza la infraestructura de DBT.
- **Single-tenant(único-inquilino):** nos permite elegir nuestra infraestructura, en la forma de instancias de un proveedor de nube (ej. Azure o AWS). Una vez desplegadas estás instancias, DBT Labs se encargará de instalar y mantener el software en ellas.

DBT Core es gratuito y de código abierto, mientras que DBT Cloud actualmente tiene tres planes de precios con diferentes capacidades: uno gratuito, uno de pago de precio fijo y otro de precio personalizado (Alrededor de 400$ cada mes por desarrollador).

DBT Cloud tiene ciertas ventajas frente a la versión Core. En primer lugar, es mucho más escalable que Core, ya que no tienes que gestionar la infraestructura. Además, Cloud cuenta con soporte técnico para la resolución de problemas, mientras que en Core se tiene que trabajar con la comunidad de usuarios y la documentación. Cloud incluye también un planificador que facilita la integración y despliegue continuo (CI/CD) en distintos entornos (ej. Desarrollo-> Pre-Producción -> Producción) del código desarrollado.

Pero en la versión cloud, dispones un menor número de bases de datos compatibles. Por el otro lado, la naturaleza de código abierto de Core permite, con el conocimiento adecuado, diseñar un sistema adaptado a las necesidades de una organización y/o un proyecto sin depender de soporte oficial. Un ejemplo de esto son los conectores con bases de datos creados por la comunidad de DBT.

En la siguiente tabla se resumen las principales diferencias entre las versiones DBT Core y Cloud. En el caso de la infraestructura, la versión cloud podemos catalogarla como Software as a Service (Software como Servicio), aunque también se ofrece la posibilidad de alojar el servicio en una infraestructura cloud de otros proveedores, como Amazon AWS o Microsoft Azure.

	Core	Cloud
Naturaleza	Código abierto	Propietario
Infraestructura	Propia/Local	SaaS
Soporte oficial	No	Si
Rendimiento	Dependiente del motor de la BD en uso	Dependiente del motor de la BD en uso
Herramientas externas	Si	Si
Desarrollo	Código	Código
Entorno de trabajo	Editor de texto de libre elección	Web de DBT Cloud

En la Figura 2, podemos ver el desarrollo con la versión Core, que requiere usar un editor de texto o, más recomendable, IDE como MS Visual Studio Code, como se muestra en la siguiente captura de ejemplo.

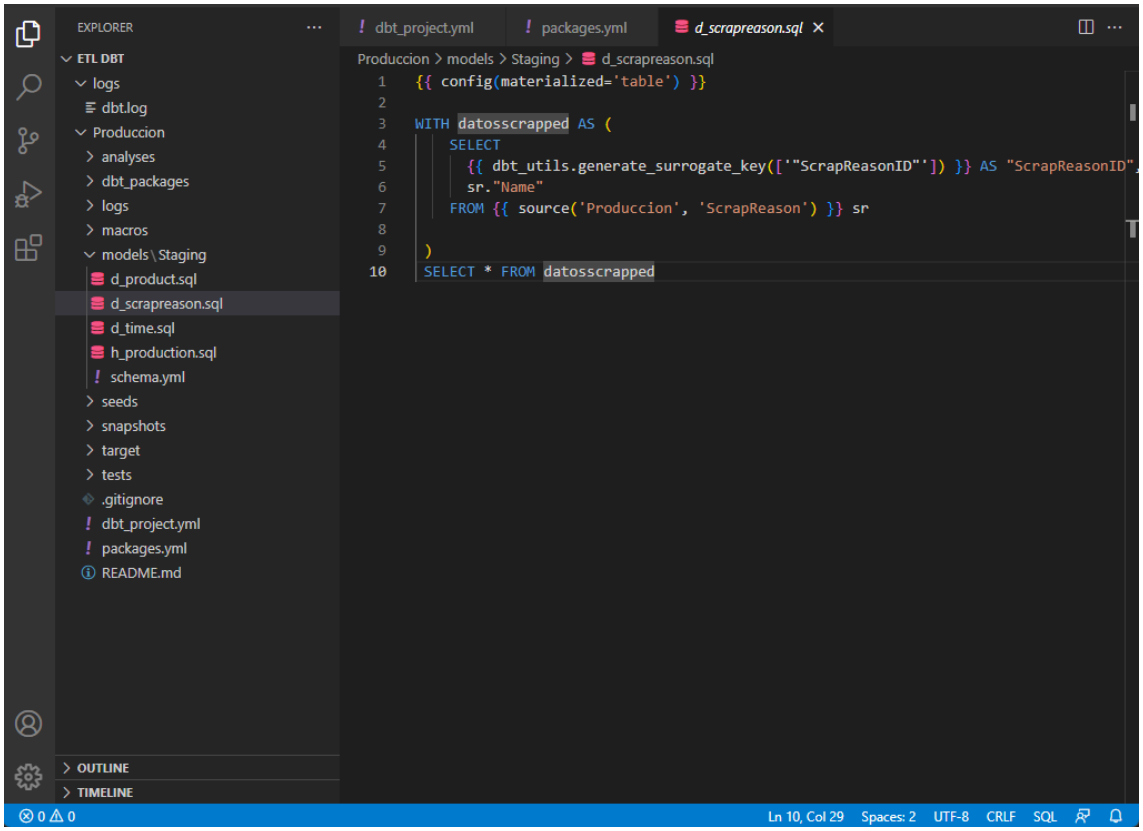


Figura 2: DBT Core

En cambio, para la versión DBT Cloud se proporciona un entorno de desarrollo web, como se muestra en la Figura 3.

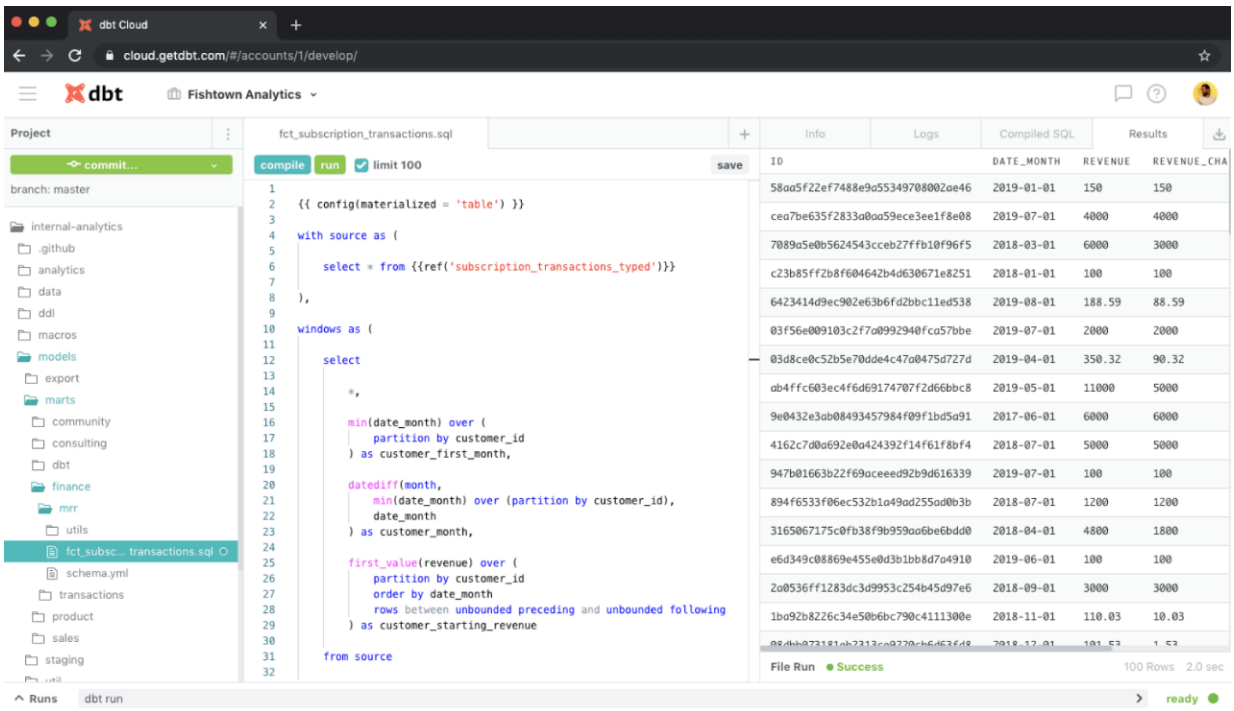


Figura 3: DBT Cloud

DAG para la representación y depuración de los procesos

El módulo de Documentación de DBT permite generar el conocido como Lineage Graph (Grafo de Linaje) tanto en DBT Core como en DBT Cloud. El Lineage Graph es un DAG interactivo generado automáticamente que permite visualizar el DAG de un modelo o de una fuente completa.

Un Grafo Acíclico Dirigido, o por sus siglas en ingles DAG, es un grafo que permite representar de forma gráfica los flujos de datos, las dependencias entre estos y, por tanto, su orden de ejecución. En la Figura 4 podemos ver un ejemplo de DAG para representar un proceso ETL en DBT.

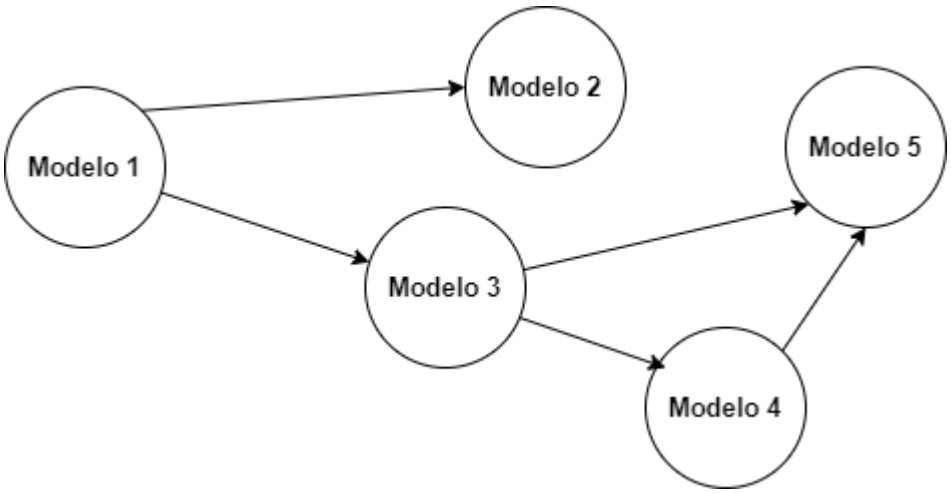


Figura 4: Ejemplo de DAG

Además, los DAG son útiles a la hora de auditar el modelo de datos porque permiten determinar modelos y relaciones ineficientes mediante la identificación de JOINS costosos, lógica compleja de datos o grandes volúmenes de datos almacenados entre otros. En la Figura 5 podemos ver un ejemplo de Lineage Graph (DAG) en DBT.

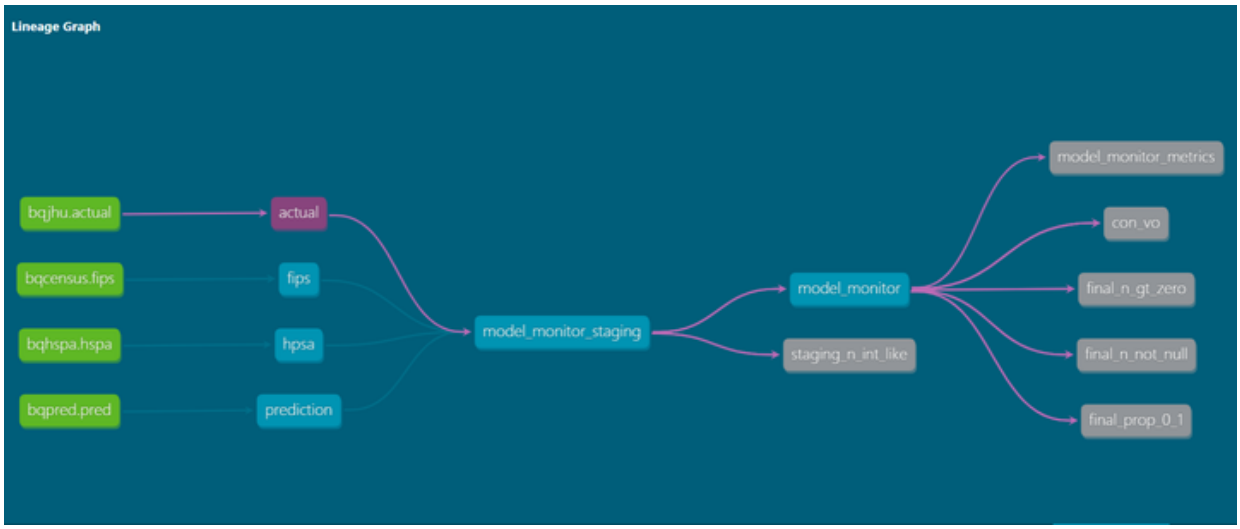


Figura 5: Ejemplo de Lineage Graph

4. CASO DE USO

Para poner a prueba DBT, hemos desarrollado un caso de uso. Utilizaremos como fuente la base de datos de Microsoft Adventure Works para este ejemplo. Esta base de datos permite almacenar y gestionar las operaciones de Adventure Works Cycles, una empresa ficticia que se dedica a la fabricación y venta de bicicletas.

El objetivo es diseñar un modelo de almacén de datos (Data Warehouse) basado en un esquema en estrella o copo de nieve para poder soportar procesos de análisis de datos y, entonces, usar DBT para transformar los datos y cargarlos en este modelo analítico objetivo.

No obstante, para nuestro objetivo de poner a prueba DBT, hemos decido hacer uso solo de algunas de las tablas fuente del modelo transaccional "Production". Las 6 tablas fuente seleccionadas se muestra en la Figura 6.

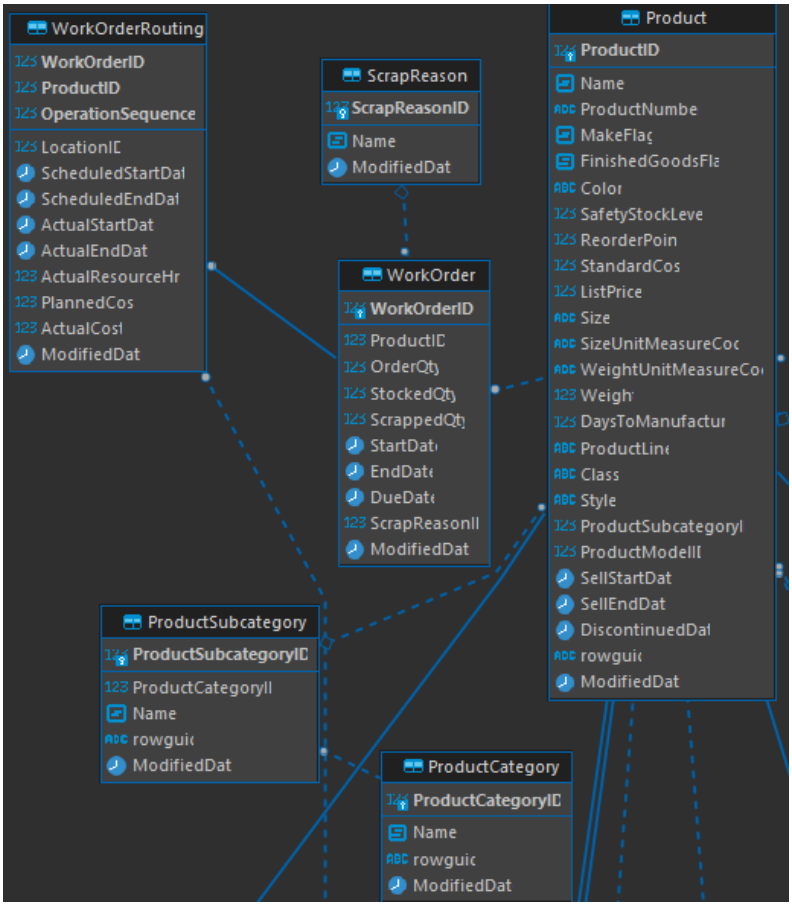


Figura 6: Tablas transaccionales

Para nuestro objetivo, hemos diseñado el siguiente modelo en estrella para la implementación del Data Warehouse. El modelo propuesto se compone de tres tablas de dimensión y una tabla de hechos:

- **"dim_product" :** contiene características analizables sobre los modelos a fabricar.
- **"dim_scrapreason" :** las razones por las que algunos productos se desechan durante el proceso de fabricación.
- **"dim_time" :** información adicional de una fecha dada. Por ejemplo: El trimestre o si es fin de semana.
- **"fact_producction" :** cada fila describe la fabricación de un producto determinado.

En la Figura 7 el diagrama del modelo en estrella propuesto.

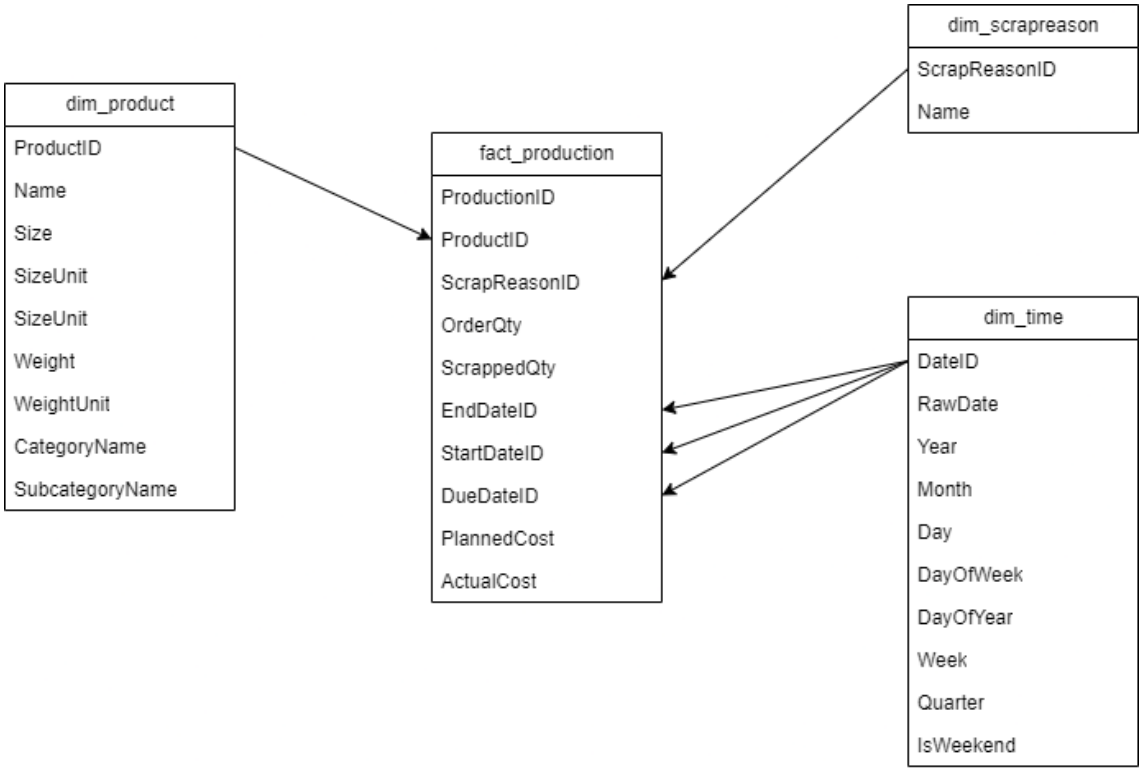


Figura 7: Modelo analítico

Decidimos utilizar la versión Core de DBT Core junto con PostgreSQL como motor de procesamiento. PostreSQL es una base de datos de código abierto con un buen rendimiento y una amplia comunidad de usuarios.

Para su implementación hemos seguido los pasos de la documentación oficial de DBT. En primer lugar, comprobamos que la instalación se ha realizado correctamente con el comando "dbt --version".

Una vez tenemos DBT en nuestra máquina, usando la consola navegamos al directorio donde alojaremos nuestro proyecto. Una vez ahí, usando del comando "dbt init <<nombre del proyecto>>", lo crearemos. Abrimos dicho directorio con Visual Studio Code, y abrimos la terminal de VS Code.

Los proyectos DBT tienen una estructura por defecto. Algunas de las carpetas más importantes son:

- **"*logs*" :** almacena los registros de ejecución.
- **"models" :** almacena los modelos.
- **"*dbt_packages*" :** listado de paquetes instalado.
- **"*macros*" :** almacén de las macros que hayamos creado.
- **"*target*" :** almacén de modelos y tests que hayamos compilado.

Las transformaciones se implementan mediante uno o más elementos de tipo modelo. En ellos escribiremos múltiples sentencias SQL, complementándolas con código Jinja en caso necesario. En estas podremos seleccionar los datos que deseamos, renombrarlos, ordenarlos y agruparlos tal y como lo haríamos en cualquier consulta SQL.

En la Figura 8 vemos un ejemplo de código Jinja, donde vemos el uso de bucles "for", condicionales, y su interacción con SQL.

```
{# Use a 'for' loop to iterate over a list of columns and generate a SELECT statement #}
{% for column in ['id', 'name', 'email'] %}
  {{ column }}{% if not loop.last %},{% endif %}
{% endfor %}
```

Figura 8: Ejemplo de Código Jinja

A continuación, en la Figura 9, mostramos la implementación en DBT de la carga de la tabla de hechos "h_production.sql".

```sql
{{ config(materialized='table') }}

WITH WorkOrder AS (
    SELECT
        wo."ProductID",
        wo."ScrapReasonID",
        wo."OrderQty",
        wo."ScrappedQty",
        t1."DateID" AS "EndDateID",
        t2."DateID" AS "StartDateID",
        t3."DateID" AS "DueDateID",
        wor."PlannedCost",
        wor."ActualCost"
    FROM {{ source('Produccion', 'WorkOrder') }} wo
    LEFT JOIN {{ source('Produccion', 'WorkOrderRouting') }} wor USING ("WorkOrderID")
    LEFT JOIN {{ ref('d_time') }} t1 on wo."EndDate" = t1."RawDate"
    LEFT JOIN {{ ref('d_time') }} t2 on wo."StartDate" = t2."RawDate"
    LEFT JOIN {{ ref('d_time') }} t3 on wo."DueDate" = t3."RawDate"
    ORDER BY "ProductionID"
)

SELECT * FROM WorkOrder
```

Figura 9: h_production.sql

Como podemos observar, gracias al lenguaje SQL podemos renombrar y reestructurar los datos fuente.

Por otro lado, hemos usado una función de Jinja en las líneas 16, 17 y 19: "{{ ref('d_time') }}" la cual hace referencia a otro modelo llamado "d_time.sql". Con este modelo nuestro objetivo es crear una tabla de dimensión tiempo.

Gracias al uso de Jinja, podemos llamar a otro modelo del proyecto en lugar de a una tabla, permitiendo crear una dependencia entre "d_produccion" y "d_time", debiéndose ejecutar el ultimo antes que el primero. De esta forma enlazamos dos nodos en nuestro DAG, como se representa en la Figura 10.

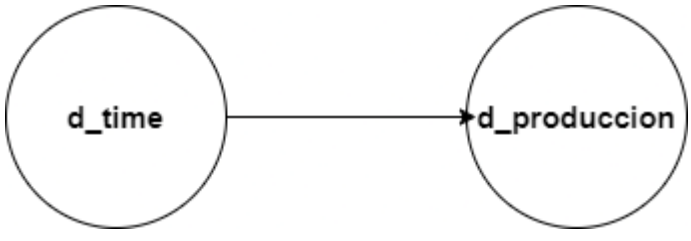


Figura 10: representación de la relación en el DAG del proyecto

En la Figura 11, se muestra la implementación del proceso de carga de la tabla de dimensión "d_time" con más detalle.

```
{{ config(materialized='table') }}

WITH tiempo AS(
    SELECT
    (date_part('year', date_day) * 10000 + date_part('month', date_day) * 100 + date_part('day', date_day)) AS "DateID",
    date_day AS "RawDate",
    date_part('year', date_day) AS "Year",
    date_part('month', date_day) AS "Month",
    date_part('day', date_day) AS "Day",
    date_part('dow', date_day) AS "DayOfWeek",
    date_part('doy', date_day) AS "DayOfYear",
    date_part('week', date_day) AS "Week",
    date_part('quarter', date_day) AS "Quarter",
    (date_part('dow', date_day) = 0 OR date_part('dow', date_day) = 6) AS "IsWeekend"
    FROM ({{  dbt_utils.date_spine(datepart="day",
                                   start_date="cast('1970-01-01' as date)",
                                   end_date="cast('2099-12-31' as date)"  )}})
)
SELECT * FROM tiempo
```

Figura 11: d_time.sql

En este modelo "d_time" que hemos creados, podemos observar un ejemplo práctico del uso de los paquetes disponibles en DBT. En este caso hemos usado "date spine" del paquete "dbt utils" que nos permite generar una tabla de dimensión tiempo que contiene más información sobre una fecha dada. Por ejemplo, podemos saber a qué trimestre pertenece una fecha o si un día es fin de semana. Los paquetes se especifican en el fichero "packages.yml" en la ruta principal del proyecto, como se muestra en la Figura 12.

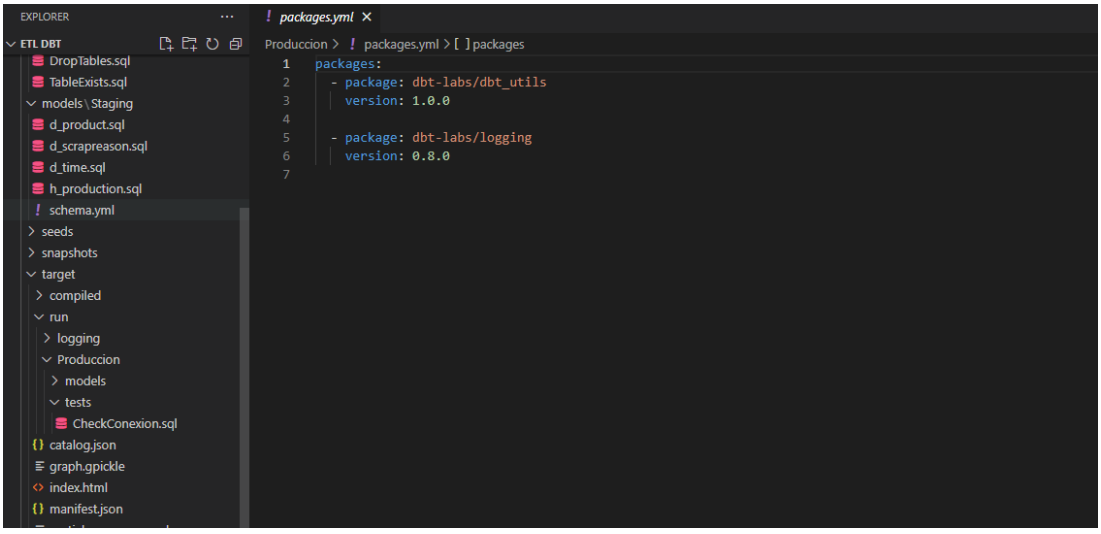


Figura 12: fichero packages.yml

En la Figura 13 se muestra el proceso de carga de la dimensión “d_scrapreason”.

```
{{ config(materialized='table') }}

WITH datosscrapped AS (
    SELECT
      "ScrapReasonID",
      sr."Name"
    FROM {{ source('Produccion', 'ScrapReason') }} sr

)
SELECT * FROM datosscrapped
```

Figura 13: d_scrapreason.sql

Para la implementación del modelo “d_scrapreason” volvemos a combinar el uso de expresiones Jinja con SQL. En esta ocasión, utilizamos la función “{{ source('Produccion', 'ScrapReason') }}”. Con esta función especificamos que el origen de los datos es la tabla “ScrapReason”, de la fuente de datos a la que hemos llamado “Produccion” en el archivo "schema.yml”.

Por último, en la Figura 14 observamos como hemos especificado el nombre de la fuente, base de datos y esquema. Las credenciales para acceder a dicha base de datos se indican en el archivo “profiles.yml” ubicado en la ruta “C:/users/<<usuario>>/.dbt” que se menciona en la documentación oficial.

```
version: 2

sources:
  - name: Produccion
    database: Produccion
    schema: Transaccional
    tables:
      - name: Product
        description: Son los productos que se fabrican
      - name: productSubcategory
        description: Detalla la subcategoria a la que pertenece cada producto
      - name: productCategory
        description: Detalla la categoria a la que pertenece cada producto
      - name: ScrapReason
        description: razon de deshecho de un producto fabricado
      - name: WorkOrder
        description: Orden de fabricacion de un producto
      - name: WorkOrderRouting
        description: Detalles de la orden de fabricacion de un producto
```

Figura 14: schema.sql

Documentación en DBT:

El fichero "schema.yml" que podemos encontrar en la carpeta "models", contiene metadatos sobre los modelos y las fuentes de datos. En este archivo podemos añadir debajo de cada fuente y/o modelo una etiqueta de descripción, las cuales se tendrán en cuenta para la generación de documentación automática. En las figuras 15 y 16 podemos ver la especificación de los modelos "d_producto" y "d_time" en "schema.yml" respectivamente.

```yaml
models:
  - name: d_producto
    description: Dimension de los productos
    columns:
      - name: ProductID
        description: Identificador del producto

      - name: DES_Producto
        description: Nombre del producto

```

Figura 15: Ejemplo de la etiqueta "description"

```yaml
  - name: d_time
    description: Dimension de tiempo
    columns:
      - name: id
        description: Identificador de la fecha

      - name: year
        description: Año de la fecha
```

Figura 16: Segundo ejemplo de la etiqueta "description"

Y así, para todas las dimensiones. Una vez lo tengamos, en la consola de Visual Studio, introducimos los siguientes comandos:

- *$ >dbt docs generate*
- *$ >dbt docs serve*

El botón azul de la esquina inferior derecha nos permite crear el DAG. En nuestro caso, las dependencias son simples, pero sirven para ilustrar el concepto, como se muestra en la Figura 18.

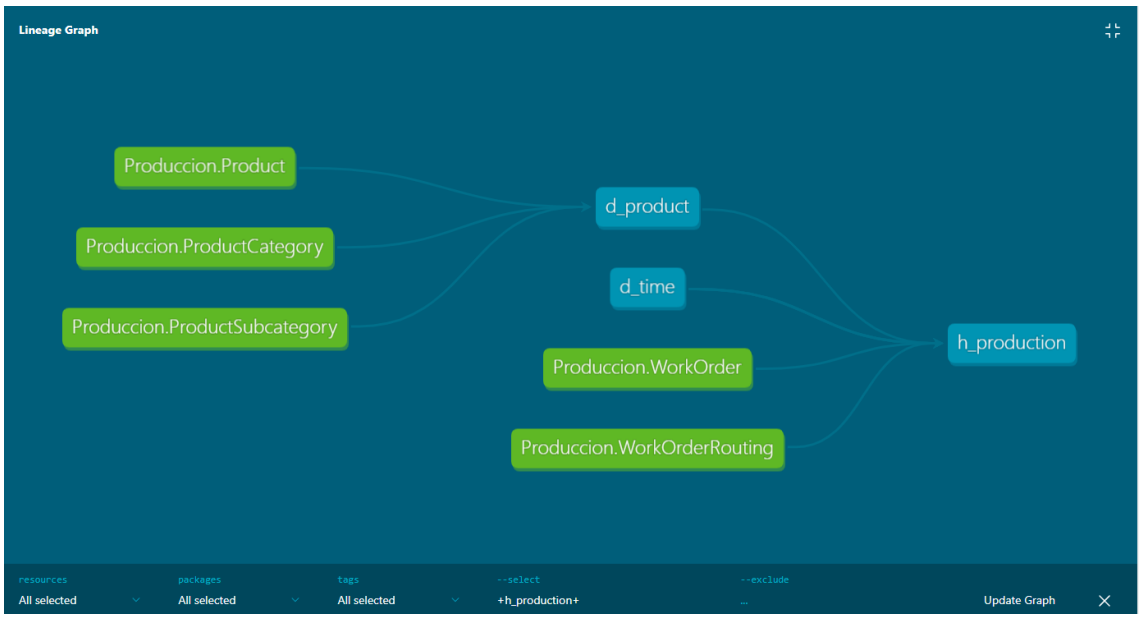


Figura 18: Lineage Graph de nuestro proyecto

Una vez tenemos todas las dimensiones y tabla de hechos, podemos utilizar el comando "$ >dbt run" para ejecutar la transformación y poder cargar las dimensiones y hechos en el esquema que hayamos especificado en "profiles.yml".

5. COMPARATIVA ENTRE PDI Y DBT

Pentaho Data Integration, más conocido como PDI, es otra herramienta ETL de código abierto. PDI tiene dos versiones: Community y Enterprise. La versión Community ofrece las funcionalidad de transformación de datos que la versión Enterprise. Esta versión Enterprise añade principalmente un planificador web de ejecución (de igual forma que DBT Cloud), control de versiones para el desarrollo y soporte oficial por parte de Hitachi.

Algunas de las similitudes entre DBT y PDI son que ambas son de código abierto (DBT en su versión Core), lo que permite ajustarlos a las necesidades de cualquier proyecto y no depender de ningún proveedor para añadir ninguna funcionalidad, ya que cada organización puede implementar cualquier mejora que vea oportuna. Ambas permiten la entrada de datos desde diferentes fuentes de datos.

En cuanto a sus diferencias, la forma de trabajo de PDI está basada en una interfaz gráfica, donde se arrastran cajas a un tapiz, se configuran mediante menús y se enlazan entre sí para representa el grafo del flujo de datos.

Sin embargo, DBT está basado en el desarrollo mediante código (SQL + Jinja/Python) y carece de una interfaz de tipo drag & drop. En el caso de DBT las configuraciones de cada paso (nodos) y relaciones (aristas o flujos) entre esos se implementan utilizando SQL y Jinja.

PDI es una herramienta completa en cuanto al proceso ETL, mientras que DBT está mucho más enfocada en la parte de la transformación (T). Además, PDI puede conectarse a una gran variedad de fuentes de datos (JDBC, CSV, JSON, HDFS,...) , mientras que DBT se conecta unicamente con bases de datos o sistemas de procesamiento que entienden el lenguaje SQL. Por estos motivos DBT normalmente será usada con otras herramientas que soporte extracción y carga (EL) como PDI o Talend, para el movimiento de datos hacia y desde el repositorio que usemos para ejecutar la transformación en DBT.

Por último, resaltar que PDI no ofrece versión Cloud, a diferencia de DBT, que ofrece una versión Cloud con planes gratuitos y de pago, dependiendo de sus funcionalidades. No obstante, existe una herramienta Cloud llamada Apache Hop basada en PDI, la cual queda fuera del ámbito de la presente comparativa.

Comparativa de rendimiento entre PDI y DBT

Una vez analizadas las diferencias y similitudes entre PDI y DBT, vamos a ver como se comparan en cuanto a rendimiento. Para intentar que las pruebas se realicen en la mayor igualdad posible, ambas herramientas están conectadas a bases de datos PostgreSQL tanto en origen como en destino, y las pruebas se realizarán en el mismo equipo. Ambos destinos no contarán con ninguna tabla creada a priori.

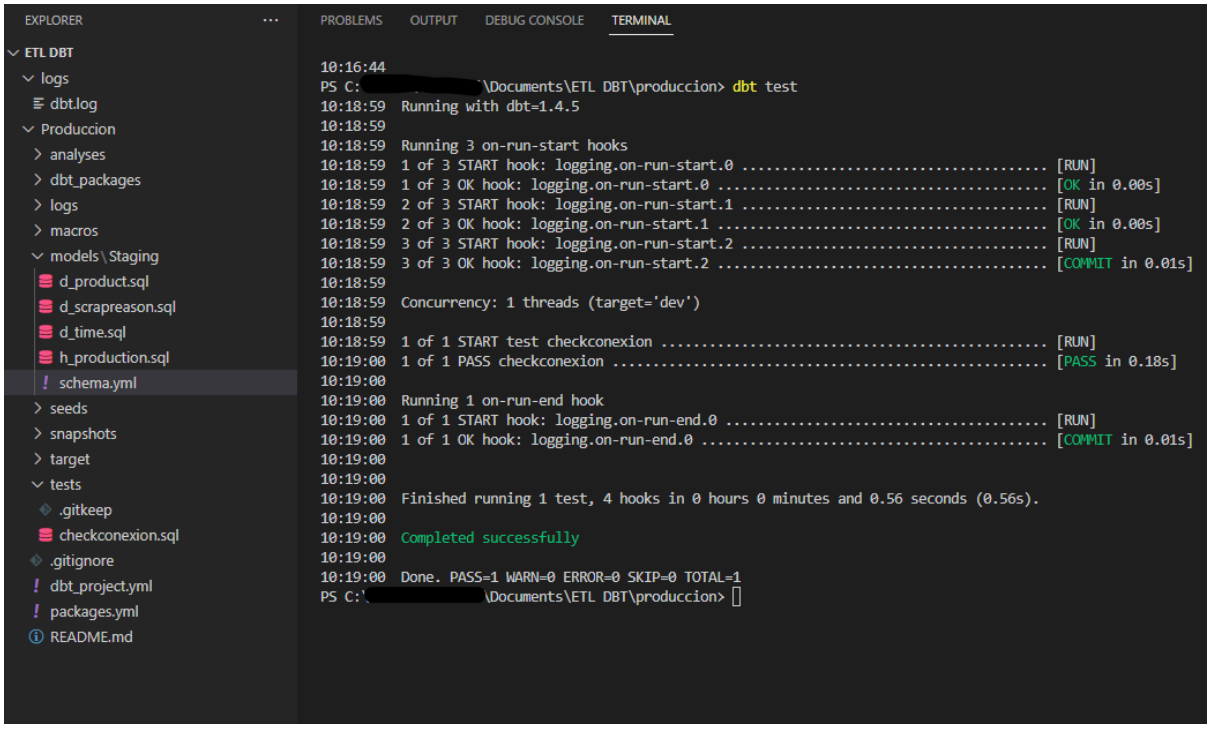


Figura 2: Ejecucion del comando "dbt test"

Para DBT, utilizaremos los comandos “dbt test” y “dbt run” en la consola de VS Code. Esto compilará y ejecutará nuestras pruebas y modelos respectivamente. DBT crea automáticamente las tablas en destino si no están creadas y no genera duplicados en tablas o en datos si ya existe. Por lo tanto, solo debemos comprobar que la conexión con la base de datos esta activa. Ejecutamos primero “dbt test” y después Y ahora “dbt run”, como se muestra en la siguiente captura.

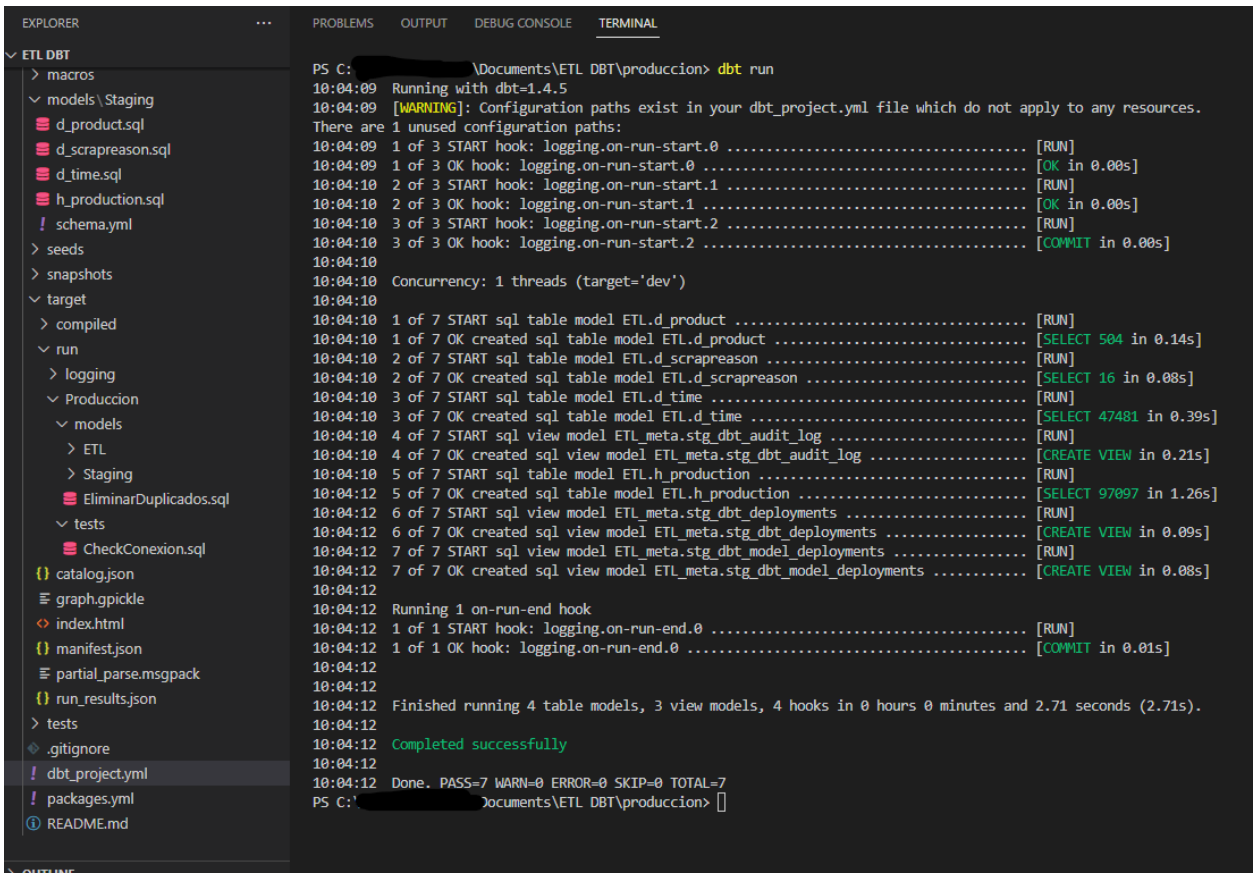


Figura 3: Ejecución del comando "dbt run"

En el caso de PDI, se puede trabajar de dos maneras: Utilizando principalmente las cajas de pasos o código SQL puro. En este caso hemos decidido seguir el proceso de construcción de trabajos utilizando cajas. Este proceso es el más común. Aunque el uso de SQL puro sea más rápido, es más complejo de depurar. Utilizaremos las cajas y sus relaciones para estructurar el trabajo.

1º) Comprobación de la estructura: En el trabajo de PDI que se representa en la Figura 21, se procede a comprobar si en el destino existe una tabla para cada dimensión y hecho. En el caso de no haber, se procede a crear una tabla con los atributos necesarios. En caso contrario, se borra dicha tabla y se vuelve a crear para evitar conflictos a la hora de cargar los datos. Así, para todas las tablas.

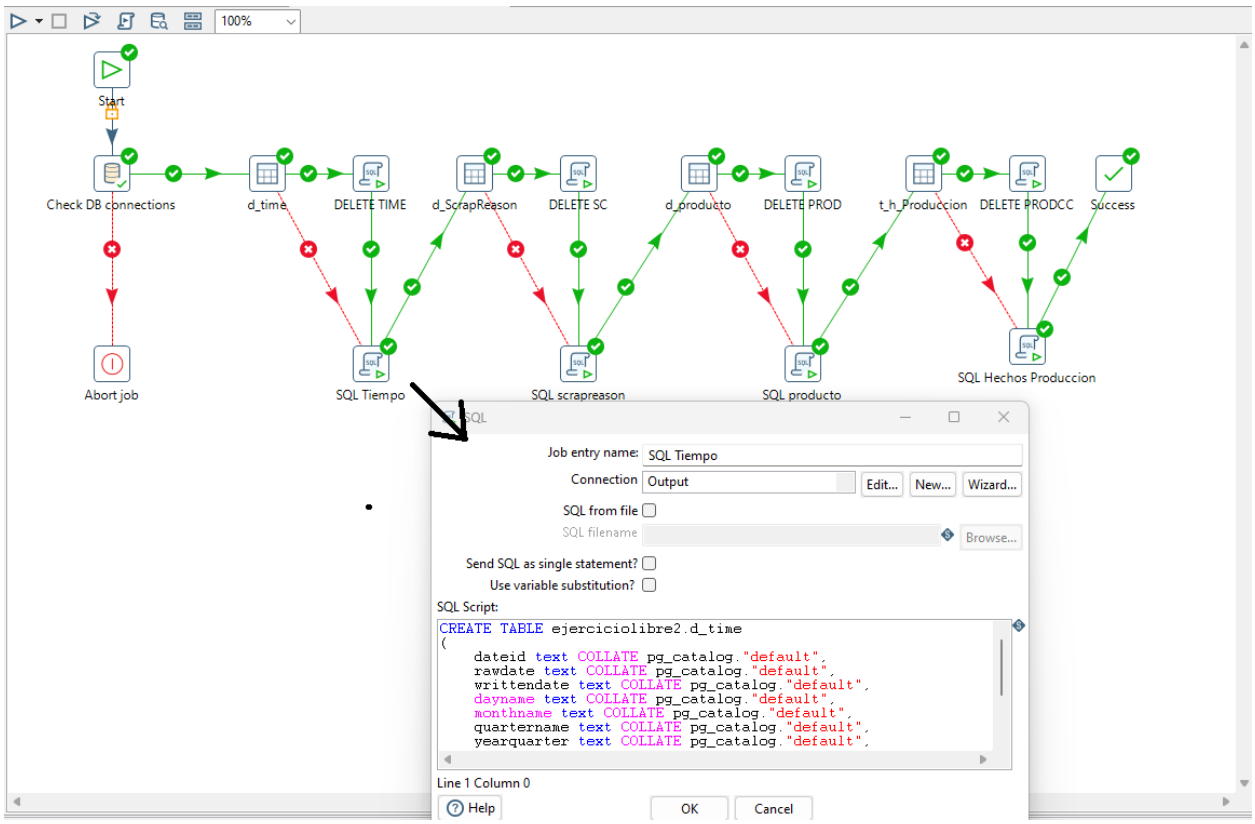


Figura 4: Proceso de comprobación de estructura

2º) Carga de datos: En trabajo de la Figura 22, vemos como se ejecutan en orden otros 3 trabajos: Primero se comprueba la estructura, tras esto se cargan las dimensiones y finalmente la tabla de hechos.

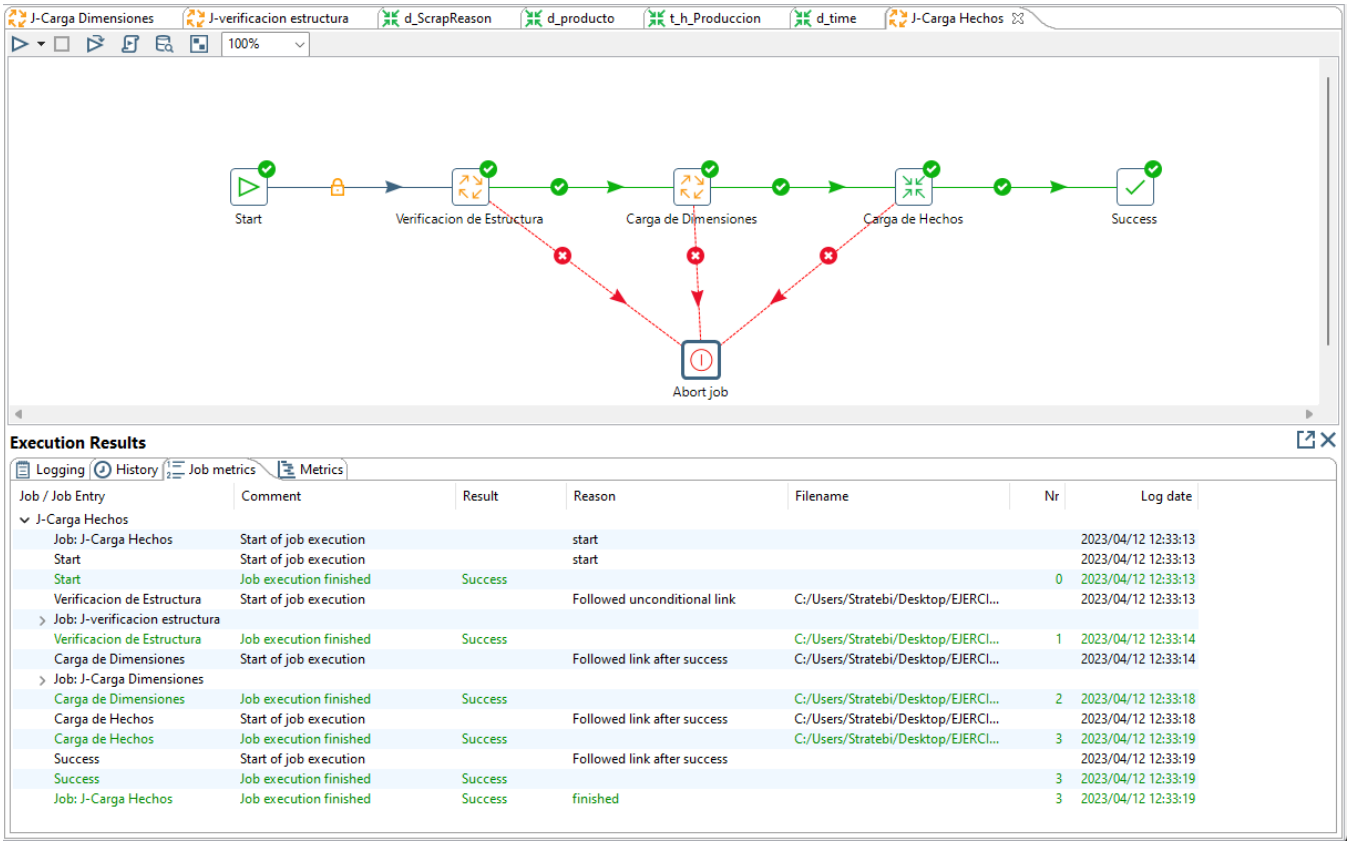


Figura 22: carga de dimensiones y hechos

Resultados de las pruebas:

Hemos realizado una prueba de tiempo de ejecución, tratando de conseguir la igualdad de condiciones entre PDI y DBT. Estos son los tiempos medios:

- DBT: 5 segundos
- PDI: 9,67 segundos

Como podemos observar, los tiempos son algo más altos en PDI aunque tampoco demasiado. Donde pensamos que DBT podría aventajar claramente a PDI es en el caso de escenarios de Big Data, con grandes volumenes de datos, y eligiendo motores de procesamiento como Spark o Databricks.

6. CONCLUSIONES

DBT una herramienta de transformación potente y fácil de usar, que emplea lenguajes populares y robustos para operar. Se integra con potentes bases de datos y motores de procesamiento Big Data para la ejecución de las transformaciones de datos y, además, es compatible con otras herramientas de ETL y orquestación del mercado que la complementan.

En nuestro estudio hemos podido poner a prueba DBT, mediante un proyecto de ejemplo. Tras esto, en la comparación con Pentaho Data Integration (PDI), hemos visto que DBT no proporciona motor de base de datos propio para ejecutar las Transformaciones de datos, si no que se integra con motores de base de datos distribuidos y, por tanto, muy potentes, como Databricks o Azure Synapse. Esto permite a DBT una mayor escalabilidad frente a otras herramientas de transformación de datos tradicionales como PDI, que ejecuta sus transformación como código Java en la misma máquina se ejecuta la herramienta.

Sin embargo, a diferencia de PDI, DBT en no ofrece una interfaz gráfica drag & drop para desarrollar los procesos de transformación y, además, no está pensada para la extracción y movimiento de datos desde y hacia el repositorio que se elija para integrar DBT. Para complementar estas carencias, DBT permite integración con algunas herramientas de ETL como Fivetran.

En resumen, DBT es una herramienta muy flexible y completa para el desarrollo de procesos de transformación de datos, que gracias a la independencia del motor de ejecución y su compatibilidad puede transformar grandes y complejos volumenes de datos en escenarios tradicionales o de tipo Big Data.

Precisamente, como trabajo futuro nos gustaría poner a prueba la integración de DBT con Apache Spark o con su versión comercial la nube Databricks para analizar dichas capacidades en escenarios de Big Data.