

Arquitectura de datos Cinemark

Gonzalo Slucki¹, Florian Reyes², and Federico Pereira³

¹gslucki@mail.utdt.edu

²freyes@mail.utdt.edu

³f.pereira@mail.utdt.edu

Julio 2024

Índice

1. Arquitectura	2
1.1. Origen de los datos	2
1.2. Uso de los datos	2
1.3. Linaje de los datos	2
1.3.1. Origen	2
1.3.2. Airflow de recolección	3
1.3.3. Airflow con DBT	3
1.3.4. Aplicaciones	3
1.4. Gobernanza de los datos	3
1.4.1. Adminstrador	3
1.4.2. Aplicaciones	3
1.4.3. Analistas	3
2. Flujo de carga de datos	3
2.1. Motivación	3
2.2. Nodos	4
2.2.1. Nodo gen_data	4
2.2.2. Nodo branch_sunday	4
2.2.3. Nodo forecast	4
2.2.4. Nodo end_sunday	5
2.2.5. Nodo warn	5
2.2.6. Nodo end_forecast	5
3. Enriquecimiento	6
4. Consideraciones	7

1. Arquitectura

1.1. Origen de los datos

En nuestro cine, los datos provienen de 4 distintas fuentes. Por un lado, las **películas** y **actores** provienen de *Hollywood*, las **salas** y los **cines** de la estructura del cine mismo, las **funciones** son generadas por el cine, y las **compras** y **clientes** vienen de los clientes mismos.

En el contexto del TP, nos enfocamos en los datos de las compras y los clientes. Para esto, usamos la biblioteca Faker, que nos permite generar e insertar los datos en nuestra base. En el contexto real, las compras y clientes llegan en una cadencia de minutos provenientes de la web propia del cine, y estimamos que lleguen aproximadamente 5.000 compras diarias en días exitosos. Sin embargo, con el fin de simplificar la carga de datos, asumiremos que llegan con cadencia diaria y que se hacen 1.000 ventas por día.

Para comenzar, generamos 1000 datos de cada tabla, con el script que se puede ver en *data_generator.py*. Luego, con frecuencia diaria y como se explicará más adelante, generamos compras y clientes.

1.2. Uso de los datos

En nuestro modelo, principalmente para los DAGs y para DBT, se hace uso de distintas partes de los datos. Para el DAG requerimos los precios de las funciones, para multiplicarlos por la cantidad de compras que se ejecutaron en esa funcion, para obtener así los beneficios generados. Es importante también tener un timestamp para poder agrupar por fecha y obtener el día, para estudiar la temporalidad de los profits y así poder hacer un forecast.

Para DBT, requerimos la información de los clientes para poder generar vistas con los clientes que más frecuentan el cine y clasificarlos acorde a su frecuencia de visitas, medida a través de las compras. Además, para nuestra segunda vista de los géneros con más dinero generado utilizamos este identificador en conjunto a la información de las funciones y de las compras, para ver qué tanto dinero logró recaudar cada género.

1.3. Linaje de los datos

1.3.1. Origen

Los datos se originan en las aplicaciones finales de la aplicación en las cuales los usuarios compran entradas y tambien el sistema central del cine en el cual se agregan las funciones.

1.3.2. Airflow de recolección

Un DAG recopila los datos nuevos de compras y los inserta en la base de datos Postgres. También los procesa para realizar predicciones y reportes.

1.3.3. Airflow con DBT

Otro DAG de Airflow procesa estos datos generados y los transforma para generar vistas materializadas, y reportes con DBT para facilitar el análisis de los datos.

1.3.4. Aplicaciones

Finalmente las aplicaciones finales vuelven a consumir todos estos datos para mantener el funcionamiento de las terminales de venta del cine.

1.4. Gobernanza de los datos

La base de datos tiene una división de roles para proteger los datos y dividir las responsabilidades debidamente:

1.4.1. Adminstrador

Es el encargado de velar por la base de datos, asegurar que no se generen inconsistencias, que las queries corran de manera rápida y es quien otorga los permisos. Tiene todos los permisos.

1.4.2. Aplicaciones

Las aplicaciones que mantienen el funcionamiento básico del cine, tienen acceso de escritura y lectura de las tablas del cine, no a las vistas materializadas.

1.4.3. Analistas

Para la generación de informes y consultas específicas a los datos, se les otorgó acceso de lectura a las tablas y vistas a los analistas de datos, no pueden escribir.

2. Flujo de carga de datos

2.1. Motivación

En términos generales, el DAG que decidimos armar tiene un proposito de forecasting, para tratar de anticipar una potencial caída en las ventas semanales. El plan de ejecucion comienza por identificar si es Domingo y si lo es, generar un forecast de la ganancia total que se espera de la semana siguiente, en base a los datos de los últimos tres meses. Con

el resultado de este forecast, se decide si mandar una alerta a los directivos del cine o no, dependiendo de si es menor a un umbral definido por el negocio.

2.2. Nodos

El flujo consta de seis nodos, dos *PythonOperator*, dos *BranchPythonOperator* y dos *EmptyOperator*, tal como se muestran en la Figura 1

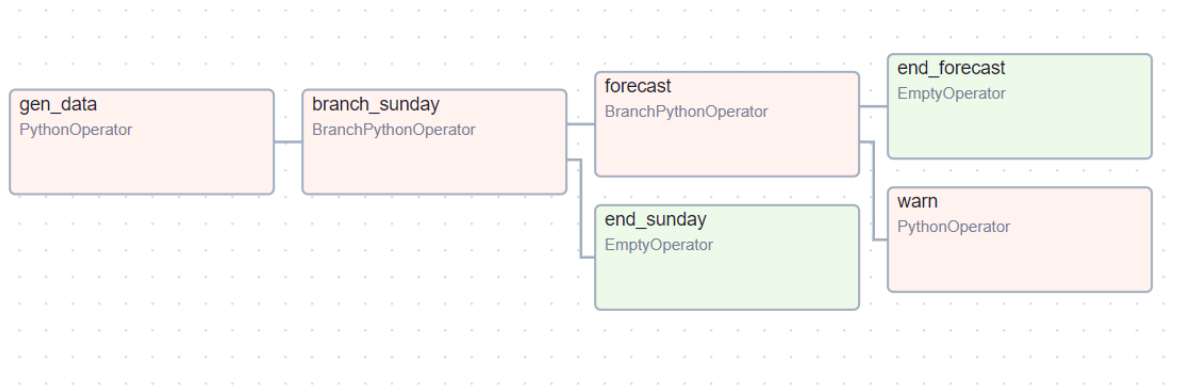


Figura 1: Flujo de cargado de datos nuevos

2.2.1. Nodo gen_data

El nodo es un *PythonOperator* que se encarga de realizar el llamado a la generación e inserción de datos. A la generación se le pasa como parámetro la n cantidad de datos a generar y se ocupa de generar esa cantidad de clientes, luego samplear n clientes para generarle a cada uno una orden de compra. Además, los inserta en la base de datos.

2.2.2. Nodo branch_sunday

Este nodo, definido como un *BranchPythonOperator* llama a una función que chequea si al momento de ejecutarse la task, es Domingo, en base a esto puede devolver un código *forecast* en caso de que sea Domingo o *end_sunday* en caso de que no lo sea, para no hacer nada.

2.2.3. Nodo forecast

Podemos definir este nodo como el corazón del DAG, acá se ejecuta el forecast basado en los datos de los últimos tres meses, lo cual se logra utilizando la librería Prophet de Python, que sirve para series de tiempo. Correr este nodo, implicó agregar una nueva query a nuestra clase Schema, la cual nos trae las recaudaciones diarias de los últimos tres meses de la base, en dos columnas, *fecha* y *profit*. Este operador también contiene lógica de branching, dado que si el forecast resultante es menor a \$5.500.000, manda una alerta a los directivos y no hace nada en caso contrario.

2.2.4. Nodo end_sunday

El nodo se ejecutará en caso de que no sea domingo y simplemente termina con el flujo. Es un *EmptyOperator*.

2.2.5. Nodo warn

En caso de que la predicción hecha por el nodo *forecast* sea menor al umbral definido, se ejecutará este nodo. Es un *PythonOperator* que hace un log de advertencia de que la ganancia esperada está debajo de ese umbral, aunque dependiendo del caso de uso podría mandar un mail o generar algún tipo de reporte. En nuestro caso decidimos que con logearlo era suficiente.

2.2.6. Nodo end_forecast

El nodo se ejecutará en caso de que no haga falta advertir de la ganancia esperada de la siguiente semana y finaliza el flujo. Es un *EmptyOperator*.

3. Enriquecimiento

Se generaron dos tablas materializadas.

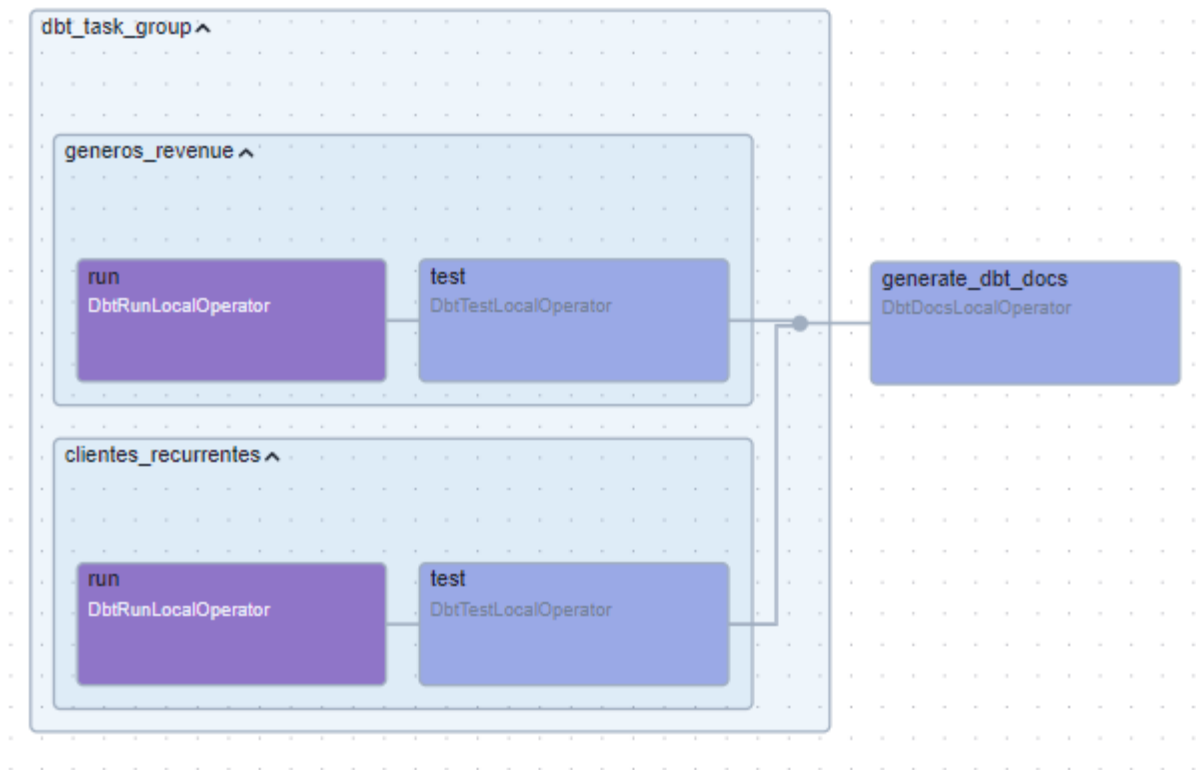


Figura 2: Flujo de cargado de dbt

Géneros revenue contiene un compilado de las ganancias del cine en el último mes diferenciados por género, esta vista se genera una vez por semana y es un dato muy importante para el cine ya que permite tomar decisiones sobre que funciones agregar para la siguiente semana teniendo en cuenta que género esta de "moda" estos dias.

Cientes recurrentes categoriza a los clientes del cine en grupos dependiendo de qué tan seguido compraron entradas. Esta es información fundamental para el equipo de marketing ya que les permite hacer promociones apuntadas a aumentar las ventas de algún grupo en específico.

Ambas tablas se actualizan una vez por semana, ya que los equipos requieren los datos de la última semana actualizados para poder continuar con sus análisis.

Luego de generar las tablas, se realiza un test a los modelos para verificar si se respetan las restricciones básicas, como que las primary keys sean únicas y que no haya valores nulos.

Se eligió como tipo de materializacion el formato de tabla, ya que no es necesario que los datos sean super frescos, con un update una vez por semana alcanza. Además a

estas tablas se les realizan muchas consultas durante la semana y nos pareció importante reducir el costo de cada consulta.






	view	table	incremental
 build time	♥ fastest — only stores logic	♥ slowest — linear to size of data	♥ medium — builds flexible portion
 build costs	♥ lowest — no data processed	♥ highest — all data processed	♥ medium — some data processed
 query costs	♥ higher — reprocess every query	♥ lower — data in warehouse	♥ lower — data in warehouse
 freshness	♥ best — up-to-the-minute of query	♥ moderate — up to most recent build	♥ moderate — up to most recent build
 complexity	♥ simple - maps to warehouse object	♥ simple - map to warehouse concept	♥ moderate - adds logical complexity

Figura 3: Comparación de tablas y view

4. Consideraciones

Si se generan demasiadas compras para una función, se puede generar un conflicto en el generador de datos, debido a la restricción de la cantidad de entradas por sala. Este es un problema de la implementación de faker, la cual entendemos no es el foco del tp. Para evitar este problemas hicimos las salas más grandes (tienen más asientos).

De la misma manera la primary key de compras la cual es incremental, la seteamos a partir del 2000 para evitar los choques con la data inicial con la que se genera la base de datos.