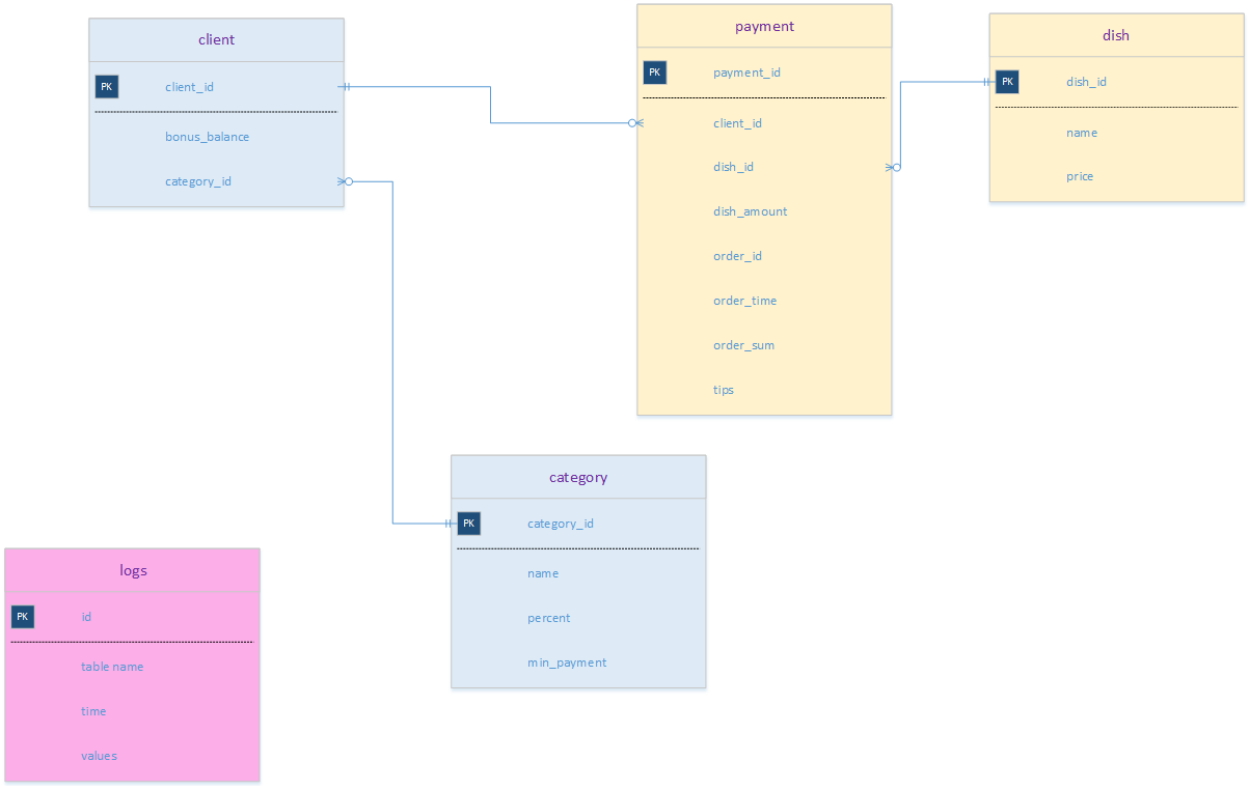
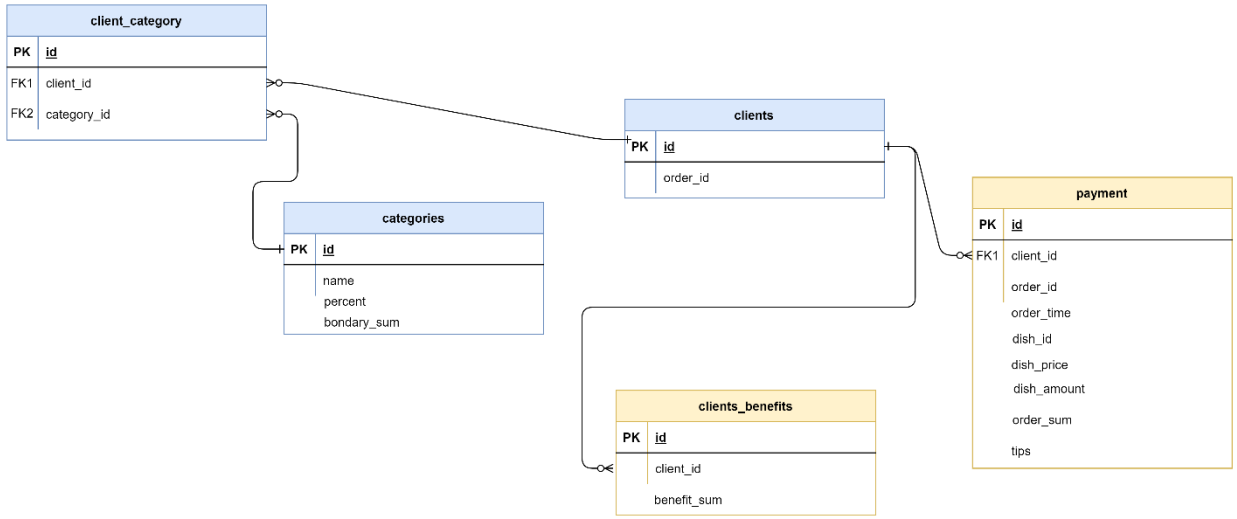
Предлагаю использовать docker compose
В PostgreSql положим
ER1
-



ER2



В «logs» сохраняются все изменения, которые внесли в другие таблицы БД
Например:

3	18.04.2024 22:40:18.143	category	{ " category_id": "1", "category_name": "gold", "percent": 0.3, "min_payment": "100 000"

			}
7	21.04.2024 04:40:18.308	order	{"order_id": 46, "client_id": 1040, …… }

В MongoDB положим три коллекции, пример на Ядиск.
Каждый документ в MongoDB содержит поле _id с типом данных ObjectId – он выполняет роль уникального ид. Два документа с одинаковыми _id вставить в MongoDB нельзя.

Orders
Restaurant
Clients

Этап Витрина

Постройте витрину с выплатами доставщикам, расчёт осуществляется за предыдущий месяц, например, в мае за апрель, где дата расчёта 21 число.

- id .первичный ключ
- deliveryman_id — id доставщика.
- deliveryman_name — фио
- year — год отчётного периода
- month — месяц отчётного периода
- orders_amount — количество заказов за месяц.
- orders_total_cost — общая стоимость заказов.
- rating — средний рейтинг доставщика клиентами.
- company_commission — сумма, которую компания забирает себе за заказ, orders_total_cost * 0.5.
- deliveryman_order_income — перечисляемые курьеру за заказ, если rating < 8, то стоимость_заказа*0,05 (но должно начислиться не меньше 400р), если rating >=10 , то стоимость_заказа*0,1 (но должно начислиться не больше 1000р)
- tips — чаевые.

Этап API

Пусть у вас есть два файла, которые надо забрать по API и догрузить в хранилище. API возвращает json и в БД надо положить json.

Можно указать параметры, например:
Параметр сортировки полей в выгружаемом файле, например, id.
Параметр для определения количества записей, которые возвращаются в качестве ответа, например, 20 или 40 или 100.
Параметр, который определяет количество записей, которые нужно забрать из полученного ответа. Например, первые 100 записей или 100 записей, начиная со 101 до 200, потом с 201 до 300 и ....
Для выгрузки доставок можно добавить два параметра, которые помогут забрать заказы с датой доставки больше или равной параметру.
Дата должна быть в формате '%Y-%m-%d %H:%M:%S’, например, “2024-05-26 00:00:00”.

Файл с доставщиками:
[{"_id":"00ga56cqwcxm789920ft8siqr ", "name":"Екатерина Великая"},
{"_id":"68ga56cqwcxm79920ft8lkjhg", "name":"Дора Величковская"}
]
Файл с доставками:
[{"order_id":"6222053d10v01cqw379td1k9",
"order_date_created":"2024-12-04 12:50:27.43000",
"delivery_id":"6222053d10v01cqw379td2t8",
"deliveryman_id":"68ga56cqwcxm79920ft8lkjhg",
"delivery_address":"Ул. Мира, 7, корпус 1, кв. 4",
"delivery_time":"2024-12-04 13:11:23.621000",
"rating":5,
"tips":500}
]

Этап проектирования

1. Опишите в файле поля, которые необходимы для витрины, а также таблицы и сущности (Mongo и API), из которых лучше взять эти поля, если поля дублируются.
2. Опишите или нарисуйте таблицы для каждого слоя хранилища, начните с витрины.

Создайте схему cdm (common Data Mart) и в ней напишите ddl – create table cdm.deliveryman_income

Создайте схему dds (detail data storage). Для этого слоя хранилища создайте снежинку. Составьте список таблиц, которые необходимо добавить в этот слой хранилища, спроектируйте структуру этих таблиц.

Например, измерение можно создать из staging.clients, из Mongo разложить json из поля obj_val в таблицу dds.dm_clients. Служебную информацию из when_updated можно не загружать. Приставку dm стоит добавить, чтобы отличать таблицы. У каждой таблицы долен быть id pk, все поля должны иметь constraint not null.

Из таблицы staging.restaurant получится dm_restaurants, но с дополнительными полями valid_from и valid_to типа timestamp. и к ней будет дополнительным измерением таблица с меню.

Создайте таблицу dm_time для измерения времени:
id
time_mark — timestamp.
year — smallint constraint year > 2022
month —smallint constraint между 1 и 12
day — smallint. constraint между 1 и 31.
time — time.
date — date.

Создайте dm_orders
Id serial

user_id — integer
restaurant_id —integer
time_id, — integer
status – varchar
order_unique_id varchar /*так надо)))*/

Добавьте новые и дополните существующие, например, в таблицу «Заказы» добавьте поле для доставщиков.
Добавьте внешние ключи, чтобы снежинка соединилась.

dm_fact_table
id – serial.
dish_id – integer.
order_id – integer.
amount – integer.
price – numeric(14, 2).
total_amount – numeric(14, 2) - это общая сумма (цена* количество единиц блюда).
bonus_payment – numeric(14, 2) поле хранит информацию о том, какую часть заказа клиент оплатил бонусами.
bonus_grant – numeric(14, 2) сколько бонусов ресторан начислил клиенту за данный заказ

Для всех полей с агрегацией:
Constraint not null
Constraint > 0

Напишите DDL для создания всех таблиц в схеме dds. Разложите json в реляционный вид.
3. Создайте схему staging. В данном слое хранилища будут лежать все данные из всех источников как есть (as is) – из Postgre, Mongo и API (json). Дополните все таблицы служебными полями – время создания/ обновления и время загрузки. В этом слое хранилища все первичные ключи, которые были serial, должны стать integer, чтобы сохранить исходные id. Можно снести все fk. Данные из Mongo можно, например, положить как

```
CREATE TABLE stg.stg_clients /*или stg.mongo_clients*/
(
    id serial CONSTRAINT stg_mongo_pk PRIMARY KEY,
    obj_id varchar NOT NULL CONSTRAINT stg_mongo_obj_id UNIQUE,
    obj_val text NOT NULL,
    when_updated timestamp /*дата и время исходного объекта из MongoDB, тогда можно отличить новые заказы от старых*/
);
```

Этап заполнение слоёв DWH

Источники данных → Слой временного хранения данных → Слой детализированных данных → Витрины
Sources (src) → Staging → DDS (Data Detail Store) → CDM (Common Data Marts)

Src	stg	dds
Mongo.clients	stg.mongo_clients	dds.dm_clients
Mongo.restaurants	stg.mongo_restaurants	dds.dm_restaurants
Mongo.orders	stg.mongo_corders	dds.dm_orders
postgre.category	stg.pg_category	dds.dm_category
postgre.dish	stg.pg_dish	dds.dm_dish
postgre.client	stg.pg_client	
postgre.logs	stg.settings	dds.settings
		dds.dm_time

Витрина в схеме cdm – cdm.deliveryman_income

Этап Airflow

Чтобы DAG появился в Airlfow, достаточно положить файлы формата .py с кодом DAG в каталог dags в Airflow. В каталоге DAG следует создать каталог ddl. В него полодить .sql файлы.
После в даг можно реализовать перебор всех файлов в каталоге ddl и выполнение скриптов. Скрипты применяются в порядке сортировки по имени файлов. Удобно когда имя файла задаётся как дата и время создания файла: yyyy-mm-dd-hhmmss-{name}. Это удобно потому, что сначала можно создать схему, а потом таблицы в схеме.
Необходимо создать несколько DAG'ов, которые наполнят все слои хранилища данными.
1. Создайте Connection в Airflow для подключения к БД источнику Postgre.
Создайте Connection для локальной инсталляции PostgreSQL, которая будет нашим хранилищем.

Для начала можно потренироваться. Создайте ваш первый даг, пусть он, например, состоит из одного шага. Он должен вычитывать полное состояние таблицы category в базе-источнике и складывать содержимое в таблицу stg.pg_category в базе DWH. Настройте его на выполнение, например, каждые 10 минут.

2. Как создать схемы с помощью DAG

```
@dag(
    # расписание выполнения дага
    schedule_interval='добавьте расписание',
    # Дата начала выполнения дага.
    start_date=pendulum.datetime(2024, 5, 24, tz="UTC"),
    # Указываем, нужно ли запускать даг за предыдущие периоды (со start_date до сегодня).
    catchup=False,
    # Задаём теги, которые используются для фильтрации в интерфейсе Airflow.
    tags=['stg', 'schema', 'ddl', 'example'],
    # В каком состоянии возникает даг (запущен / остановлен).
    is_paused_upon_creation=False  # запущен.
)
def example_stg_init_schema_dag():
    # подключение к базе DWH.
    dwh_pg_connect = ConnectionBuilder.pg_conn("____")

    # путь до каталога с SQL-файлами из переменных Airflow.
    ddl_path = Variable.get("_____")

    # Объявляем задачу, которая создаёт структуру таблиц.
    @task(task_id="schema_init")
    def schema_init():
        rest_loader = SchemaDdl(dwh_pg_connect, log)
        rest_loader.init_schema(ddl_path)

    # Инициализируем объявленные таски.
    init_schema = schema_init()

    # Задаём последовательность выполнения: пока только инициализация схемы.
    init_schema

# Вызываем функцию, описывающую DAG.
stg_init_schema_dag = example_stg_init_schema_dag()
```

3. Предлагаю всю бизнес-логику разложить по файлам в том же каталоге, и по мере вызова она будет импортироваться в DAG для вызова. Вы

можете делить написанный код на файлы, чтобы не писать всю логику в одном месте.

4. Ещё один пример с таблицей category. Файл stg_pg_category.py Можно использовать созданное подключение к базе источнику

```
@dag(
    # расписание выполнения дага.
    schedule_interval='* * * * *',
    # дата начала выполнения дага.
    start_date=pendulum.datetime(2024, 5, 24, tz="UTC"),
    # нужно ли запускать даг за предыдущие периоды (со start_date до сегодня).
    catchup=False,
    # Задаём теги, которые используются для фильтрации в интерфейсе Airflow.
    tags=['stg', 'source', 'example'],
    # В каком состоянии рождается даг (запущен / остановлен).
    is_paused_upon_creation=False  # запущен.
)
def example_stg_pg_category_dag():
    # подключение к базе DWH.
    dwh_pg_connect = ConnectionBuilder.pg_conn("____")
    # Создаём подключение к базе stg.
    origin_pg_connect = ConnectionBuilder.pg_conn("____")

    # Объявляем задачу, которая загружает данные.
    @task(task_id="categories_load")
    def load_categories():
        # Создаём экземпляр класса и тут реализуем логику.
        rest_loader = RankLoader(origin_pg_connect, dwh_pg_connect, log)
        rest_loader.load_categories()  # Вызываем функцию, которая перельёт данные.

    # Инициализируем объявленные задачи.
    categories_dict = load_categories()

    # задаём последовательность выполнения задач.
    # Задача одна, поэтому просто обозначим её здесь.
    categories_dict

stg_pg_categories_dag = example_stg_pg_categories_dag()
```

5. Должен быть ещё файлик с бизнес-логикой — categories_loader.py:

Курсор для инкрементальной загрузки

Чтобы организовать инкремент используем таблицу stg.settings. В нашей служебной таблице будем хранить положение курсора — отметку о том, какие объекты уже были перенесены в stg слой.

Сами настройки записываем в поле settings с типом JSON. Тут понадобится функция преобразования объекта в JSON- строку. Такая функция понадобится и при сохранении данных из MongoDB.

Внесение изменений в рамках транзакции

Сохранение состоит из двух частей:

- сохранение самих данных и
- сохранение курсора.

Сохранение обеих частей должно происходить одновременно. Чтобы избежать потеря данных, например, когда данные сохранили, но курсор не переместили: при следующем запуске DAG будет считает тот же набор данных. Либо курсор подвинули, а данные остались незаписанными.

Дальше грузим таблицу Clients, для этого можно расширить созданный DAG. Например, можно добавить в DAG ещё один шаг, который выгрузит полностью таблицу Clients из базы sources и переливает в таблицу stg.pg_clients.

6. Загрузим данные из postgre.logs.

postgre.logs – буфер изменений в БД источнике. В эту таблицу дописываем новые данные, а устаревшие удаляем. При каждом запуске необходимо забирать все строки.

Создадим отдельный периодический процесс.

Чтобы избежать повторной выгрузки данных при повторяющихся запусках DAG, создадим таблицу для сохранения текущего прогресса процесса.

У вас уже заведена таблица для хранения текущего курсора загруженных данных, она называется stg.settings. Таблица состоит из таких полей:

- id ид.
- settings_key — ключ(и) задачи, которая сохраняет свою отметку о загрузке данных.
- settings — произвольный набор параметров. Структура может быть отличаться для каждой задачи, поэтому JSON.

7. DAG для таблицы логов

В созданный DAG можно добавить ещё одну операци — event_load.

Эта операция считывает текущее состояние загрузки — в таблице logs по id. Используя id как метки состояния.

Например:

1. Посмотрели максимальный stg.settings.id, загруженный в прошлый раз.
2. Забрали данные из logs, где id больше, чем сохранённый
3. Сохранили последний записанный id в таблицу stg.settings.

Для MongoDB

Чтобы забрать данные можно использовать, например, пакет pymongo. В staging-слой положим данные as is.
Создаём ещё одно соединение с помощью Admin -> Variables.
Все объекты в базе имеют поле update_time. Значение в этом поле это дата и время обновления документа. Будем и дальше использовать таблицу stg.settings, для нового Mongo DAG тоже будем хранить признак, когда данные были перенесены в последний раз.
Тут есть два способа:

- Можно завести одну настройку на весь DAG или
- сохранять отдельно значения под каждую таблицу.

DAG должен перекладывать данные из всех коллекций MongoDB в уже созданный staging-слоя.
Пример дага для монги, который можно не переносить, так как мы забираем даные про рестораны из постгре

```
# order_system_restaurants_dag.py
@dag(
    #  расписание
    schedule_interval='* * * * *',
    # дата начала выполнения DAG
    start_date=pendulum.datetime(2024, 5, 24, tz="UTC"),
    # нужно ли запускать даг за предыдущие периоды (со start_date до сегодня).
    catchup=False,
    # Задаём теги, которые используются для фильтрации в интерфейсе Airflow.
    tags=['example', 'stg', 'source'],
        # остановлен DAG или запущен при создании.
    is_paused_upon_creation=False  # запущен.
)
def example_stg_mongo_restaurants():
    #  подключение к базе DWH.
    dwh_pg_connect = ConnectionBuilder.pg_conn("_____")

    # обращаемся к переменным Airflow.
    cert_path = Variable.get("MONGO_DB_CERTIFICATE_PATH")
    db_user = Variable.get("MONGO_DB_USER")
    db_pw = Variable.get("MONGO_DB_PASSWORD")
    rs = Variable.get("MONGO_DB_REPLICA_SET")
    db = Variable.get("MONGO_DB_DATABASE_NAME")
    host = Variable.get("MONGO_DB_HOST")

    @task()
    def load_restaurants():
```

```
        #  класс для сохраниние.
        pg_saver = PgSaver()

        # подключение к MongoDB.
        mongo_connect = MongoConnect(cert_path, db_user, db_pw, host, rs, db, db)

        # класс для чтения данных
        collection_reader = RestaurantReader(mongo_connect)

        # класс для загрузки данных
        loader = RestaurantLoader(collection_reader, dwh_pg_connect, pg_saver, log)

        # копирование
        loader.run_copy()

    restaurant_loader = load_restaurants()

    #порядок выполнения. пока тут только одна задача, и мы не прописываем зависимости между задачами.
    restaurant_loader

mongo_stg_dag = example_stg_mongo_restaurants()
```

Алгоритм может быть, например:

- Забираем batch данных из коллекции, например, 100.
- Преобразовываем данные в JSON.
- Сохраняем в stg.mongo_restaurants.
- Сохраняем update_time последнего сохранённого объекта в stg.settings.

Summary про DAG для заполнения STG-слоя

DAG, который будет заполнять таблицы staging-слоя данными из API.

API отдаёт данные частями. Чтобы выгрузить все данные по API, предлагаю использовать постраничное чтение (paging). Для реализации постраничного чтения, можно использовать поля sort_field, sort_direction, portion/offset , limit. Например, за первое обращение можно выгрузить первые 100 записей, при этом смещение будет 0. За второе обращение, limit можно оставить 100, а смещение будет 100, чтобы уже не читать первые 100 записей, мы их уже забрали. Далее limit 100, а смещение уже 200. Например, сощдайте выгрузку на последние 7 месяцев, 1 раз в месяц.

заполнение DDS-слоя

Создайте новый DAG для заполнения таблиц-измерений.

Создайте ещё один DAG для заполнения таблицы фактов.

Как и для stg слоя понадобится таблица с настройками - dds.settings .

заполнение CDM-слоя

Создайте DAG для заполнения витрины.

При написании запроса стоит учесть:
При вычислении показателей необходимо выбирать заказы с итоговым статусом succsess/finished/delivered. (давайте выберем, чтобы у всех он был одинаковый)
DAG запускается периодически, при повторных запусках логика не должна сломаться.
Данные в слое DDS постоянно добавляются.