Построение пайплайна в Airflow для ежедневной обработки данных сервиса такси.

- Автор: Егорова Ольга

Цели и задачи проекта

Будем работать с данными сервиса заказа такси. Для того, чтобы финансовая и продуктовая команды регулярно получали актуальные данные о выручке и поведении пассажиров, необходимо настроить автоматическую обработку данных.

Итак, нам нужно построить витрину данных, которая будет агрегировать информацию о поездках по каждому способу оплаты. Для этого напишем PySpark-скрипт, который рассчитает ключевые показатели: количество поездок, среднюю стоимость поездки, средние чаевые и суммарную выручку по каждому типу оплаты. А чтобы процесс был полностью автоматическим, создадим DAG в Airflow, который будет ежедневно:

- проверять наличие нового файла с данными;
- запускать Spark-задачу;
- формировать агрегированную таблицу.
- записывать полученнные данные в ClickHouse, они, в свою очередь, в дальнейшем станут основой для финансовых отчётов и аналитических дашбордов.

Описание данных

Таблица `taxi_data` содержит данные об активности пользователей, состоит из множества полей, но нас интересуют только 4 следующих поля:

- `fare` — стоимость поездки;
- `tips` — размер чаевых;
- `trip_total` — общая стоимость поездки;
- `payment_type` — способ оплаты.

1. ClickHouse

Первое, что нужно сделать - это создать пустую таблицу для записи данных.

Идём в DBeaver и подключаемся к ClickHouse. Для подключения потребуются следующие параметры:

- `'dbname'` — название базы данных;
- `'host'` — адрес сервера;
- `'password'` — пароль;
- `'port'` — порт;
- `'user'` — пользователь.

С помощью SQL-запроса создаём пустую таблицу для будущего агрегата, назовем её `taxi_payment_summary`

```
CREATE TABLE taxi_payment_summary (
    trip_date Date,
    payment_type String,
    trip_count UInt64,
    fare_avg Float64,
    tips_avg Float64,
    trip_total_sum Float64
)
ENGINE = MergeTree()
ORDER BY (trip_date, payment_type)
PARTITION BY toYYYYMM(trip_date);
```

2. Spark-код

Следующим шагом готовим Spark-код. Этот скрипт запишем в файл с названием `my_spark_job.py`.
Полный путь к этому файлу в нашем случае будет выглядеть так: `s3a://corporation_data/{user}/jobs/my_spark_job.py`, где `user='имя пользователя'`

▫В S3 каждый день появляется файл с данными и он всегда называется одинаково `taxi_data.parquet`. Данные хранятся в формате Parquet, поэтому для чтения используем метод `spark.read.parquet()`.

▫Полученные данные необходимо агрегировать. Сгруппируем данные по полю `payment_type` и рассчитаем четыре показателя:

- количество поездок;
- среднюю стоимость;
- средние чаевые;
- суммарную выручку.

▫И настроить запись полученной таблицы в ClickHouse.

```
from pyspark.sql import SparkSession
import pyspark.sql.functions as F

# Создаём Spark-сессию, приложение назовем `TaxiPaymentSummary`
spark = (SparkSession.builder
         .appName("TaxiPaymentSummary")
         .config("fs.s3a.endpoint", "storage.yandexcloud.net")
         .getOrCreate()
        )

parquet_path = "s3a://corporation_data/taxi/taxi_data.parquet"

# Читаем Parquet-файл
df_from_parquet = spark.read.parquet(parquet_path)


# Указываем порт и параметры кластера ClickHouse
jdbcPort = 0000
jdbcHostname = "..адрес сервера.."
jdbcDatabase = "..название базы данных.."
jdbcUrl = f"jdbc:clickhouse://{jdbcHostname}:{jdbcPort}/{jdbcDatabase}?ssl=true"


# Строим агрегат по способу оплаты
taxi_payment_summary = df_from_parquet.groupBy("payment_type").agg(
    F.current_date().alias("trip_date"),
    F.count("payment_type").alias("trip_count"),
    F.avg("fare").alias("fare_avg"),
    F.avg("tips").alias("tips_avg"),
    F.sum("trip_total").alias("trip_total_sum")
)

# Результат агрегации записываем в таблицу ClickHouse с помощью JDBC-подключения
(taxi_payment_summary.write
     .format("jdbc")
     .option("url", jdbcUrl)
     .option("user", "..имя пользователя..")
     .option("password", "..пароль..")
     .option("dbtable", "taxi_payment_summary")
     .mode("append")
     .save()
)
spark.stop()
```

3. Создание DAG

Когда Spark-код готов, нужно создать DAG, который будет его запускать.

Зададим параметры для DAG:

▫ Обязательно укажем уникальный идентификатор (`dag_id`)
▫ DAG будет запускаться каждый день (здесь используем псевдоним `@daily`)
▫ Укажем значения по умолчанию (`default_args={'depends_on_past':False}`), то есть попытки независимы и не зависят от вчерашних неудач.
▫ Обязательно укажем начало запуска, пусть это будет 1 января 2025 года (`start_date=datetime(2025, 1, 1)`)
▫ При этом запускать DAG за пропущенные даты не будем, пусть он начнет работать с текущего момента (`catchup=False`)

Добавим проверку входного файла - DAG не должен стартовать, пока в S3 не появится файл. Здесь будем использовать сенсор S3KeySensor:

- Обязательно укажем уникальный идентификатор задачи в DAG (`task_id`)
- Укажем имя бакета, в котором нужно проверить наличие файла (в данном случае это `bucket_name='corporation_data'`)
- Укажем путь к файлу, который нужно найти.
- Пусть сенсор ищет точное совпадение с bucket_key, то есть нам нужен только указанный файл (`wildcard_match-False`).
- Укажем идентификатор подключения (`aws_conn_id`) к S3 в Airflow, в котором хранятся ключи и данные для доступа к хранилищу.
- Укажем как часто сенсор будет проверять наличие файла. Сделаем каждые 5 минут (`poke_interval`)
- Укажем максимальное время ожидания 24 часа. (`timeout`)
- Пусть сенсор остается активным в памяти и проверяет условие через заданный интервал (`mode='poke'`)

➕

Создадим задачу:

Чтобы запускать Spark-приложения на кластере прямо из Airflow, с помощью оператора `DataprocCreatePysparkJobOperator`

- Обязательно укажем уникальный идентификатор задачи в Airflow `task_id`
- Обязательно укажем уникальное имя задания Dataproc `name`
- Укажем путь к файлу Spark-приложения `main_python_file_uri`
- Укажем на каком кластере запускать задание `cluster_id`

➕

И обязательно указываем зависимости

```
#Импортируем модуль для работы с датой и временем
from datetime import datetime
# Импортируем класс DAG
from airflow import DAG
# Импортируем сенсор (для новых версий Airflow)
from airflow.providers.amazon.aws.sensors.s3 import S3KeySensor
#Импортируем оператор
from airflow.providers.yandex.operators.dataproc import DataprocCreatePysparkJobOperator


DAG_ID = "my_dag_taxi_payment_summary"

with DAG(
    # Указываем необходимые параметры:
    #задаем уникальный идентификатор DAG
    dag_id=DAG_ID,
    #задаем расписание запусков -  каждый день
    schedule_interval="@daily",
    #задаем дату старта с 1 января 2025 года
    start_date=datetime(2025, 1, 1),
    #независимость
    default_args={'depends_on_past':False},
    #указываемать теги графа
    tags=["taxi"],
    #режим генерации пропущенных интервалов отключаем
    catchup=False
) as dag:
    # Создаем экземпляр сенсора, который будет ждать появления входного файла в S3
    wait_for_input = S3KeySensor(
        #задаем уникальный идентификатор задачи в DAG
        task_id='my_task_sensor',
        #проверка каждые 5 минут, то есть 300 сек.
        poke_interval=300,
        #задаем время ожидания 1 час, то есть 3600 сек.
        timeout=86400,
        #указываем имя бакета, в котором нужно проверить наличие файла
        bucket_name='corporation_data',
        #указываем путь к файлу, который нужно найти
        bucket_key= "taxi/taxi_data.parquet",
        #указываем режим работы сенсора - активен все время
        mode='poke',
        # указываем идентификатор подключения к S3 в Airflow
        aws_conn_id='s3',
        # ищем точное совпадение с bucket_key
        wildcard_match=False
    )

    # Запускаем PySpark-задание на кластере Dataproc
    user = '..имя пользователя..'
```

```
    run_pyspark = DataprocCreatePysparkJobOperator(
        # уникальное имя для задания Dataproc
        name="create_aggregate_and_load",
        # идентификатор задачи в Airflow
        task_id="my_task_create",
        #идентификатор кластера Dataproc
        cluster_id="********************",
        #указываем путь к PySpark-скрипту
        main_python_file_uri=f"s3a://corporation_data/{user}/jobs/my_spark_job.py"
    )

    # Указываем зависимости
    wait_for_input >> run_pyspark
```

5. Результат

Когда DAG вместе с задачами отработает без ошибок, убедимся, что данные успешно загрузились в ClickHouse.

Для этого выполним в DBeaver запрос:

```
SELECT *
FROM taxi_payment_summary
LIMIT 100;
```

Мы должны увидеть агрегированные данные, подготовленные в Spark-приложении.

Результат выполнения запроса - несколько строк из таблицы с данными:

```
trip_date |payment_type|trip_count|fare_avg          |tips_avg            |trip_total_sum    |
----------+------------+----------+------------------+--------------------+------------------+
2025-10-06|Credit Card |1108843   |15.869293343229563|   3.477157157788922|23160130.960003342|
2025-10-06|No Charge   |12843     |13.535953437670331|  0.1256941524565911|184374.59000000003|
2025-10-06|Unknown     |5180      | 11.79162804171493| 0.35713595983005136| 67725.07000000005|
2025-10-06|Prcard      |968       |11.172262396694215| 0.25764462809917354|11513.149999999998|
2025-10-06|Cash        |1410155   |11.194248598877067|0.003325194540318936|16971150.869999863|
2025-10-06|Dispute     |1842      |13.327953311617806|                 0.0|          27052.29|
2025-10-06|Way2ride    |3         |               3.5|  0.7933333333333333|        14.9999999|
2025-10-06|Pcard       |878       | 9.833428246013668| 0.18314350797266515|           8956.55|
2025-10-07|Credit Card |1205483   |16.541449922571126|     3.3571359598300|    22962190.96342|
2025-10-07|Unknown     |5440      | 11.35584154514523|        0.6971359598| 68836.00005000005|
2025-10-07|Prcard      |1501      |11.521453682442102|    0.33251945403115|13.999999998989898|
2025-10-07|Cash        |1652135   |11.105215448841541|        0.0540318936|18961365.986351987|
2025-10-07|Dispute     |1645      |13.951454132253689|             0.05699|   27259.300000009|
2025-10-07|Way2ride    |10        |            3.4848|     0.3333333333333|14.999999999999999|
2025-10-07|Pcard       |983       |    8.952216845151|     0.4350797266515|       8932.549811|
2025-10-07|No Charge   |13596     |12.569331598978797|    0.96891524565911|      1942224.5903|
```

Теперь ежедневно таблица будет наполняться актуальными данными, которые можно использовать для финансовых отчётов и аналитических дашбордов.