UNICHECK

Ім'я користувача:
приховано налаштуваннями конфіденційності

ID перевірки:
1015591312

Дата перевірки:
13.06.2023 20:23:48 EEST

Тип перевірки:
Doc vs Library

Дата звіту:
13.06.2023 20:24:29 EEST

ID користувача:
100011372

Назва документа: **Vitiv 1-3**

Кількість сторінок: **33** Кількість слів: **5960** Кількість символів: **43757** Розмір файлу: **1.81 MB** ID файлу: **1015240495**

0.87%
Схожість

Найбільша схожість: **0.17%** з джерелом з Бібліотеки (ID файлу: **1015015061**)

Пошук збігів з Інтернетом не проводився

0.87% Джерела з Бібліотеки | **8** ..... Сторінка 35

0% Цитат

Вилучення цитат вимкнене

Вилучення списку бібліографічних посилань вимкнене

0%
Вилучень

Немає вилучених джерел

Модифікації

Виявлено модифікації тексту. Детальна інформація доступна в онлайн-звіті.

Замінені символи | **1**

1 ТЕОРЕТИЧНІ ВІДОМОСТІ ПРИ РОБОТІ З ТЕХНОЛОГІЄЮ APACHE AIRFLOW

1.1 Переваги Apache Airflow

Apache Airflow - це платформа, призначена для програмування та планування складних обчислювальних процесів. Вона забезпечує можливість оркестрування процесів великої обробки даних, які можуть бути складними та довгими. Airflow дозволяє користувачам програмувати, планувати та моніторити обчислювальні процеси, використовуючи Python.

Apache Airflow забезпечує підтримку багатьох відкритих та пропрієтарних систем, таких як Hadoop, Apache Spark, Amazon Web Services та інші. Airflow дозволяє користувачам створювати зручні та гнучкі процеси, що забезпечують можливість масштабування та збільшення продуктивності.

Основна перевага Apache Airflow полягає в тому, що вона дозволяє відстежувати та моніторити стан обчислювальних процесів в реальному часі. Airflow надає можливість моніторингу, який дозволяє отримувати повідомлення про стан процесів та повідомляти про можливі помилки, що дозволяє зменшити витрати часу та зусиль на розв'язання проблем під час розробки проектів з обробки даних.

Apache Airflow також забезпечує гнучкість та масштабованість, що дозволяє користувачам змінювати та налаштовувати процеси в залежності від потреб проекту. Airflow підтримує декілька режимів роботи, такі як стандартний, з підтримкою паралельної обробки, з підтримкою розподіленої обробки, що дозволяє вибрати оптимальний режим роботи для проекту з обробки даних.

У загальному, Apache Airflow - це потужний інструмент для оркестрації та автоматизації великих обчислювальних процесів. Він забезпечує можливість програмування, планування та моніторингу процесів з обробки даних з використанням зручного і простого інтерфейсу, який базується на мові Python. Це дозволяє розробникам та аналітикам зосередитися на роботі з даними, замість

витрачання часу на написання складних скриптів для оркестрування процесів.

1.2 Архітектура Apache Airflow

Архітектура Apache Airflow складається з трьох основних компонентів:

1. Сервер Airflow (Airflow Server): це основний компонент, який керує процесом оркестрування. Він відповідає за розподіл завдань та їх виконання, керування ресурсами, налаштування з'єднань з базами даних та зовнішніми сервісами, моніторинг та логування процесів.
2. База даних (Metadata Database): це база даних, в якій зберігаються всі дані про процеси, що виконуються в Apache Airflow. Це включає дані про задачі, DAG-и, розклади запуску та інші метадані.
3. Робітники (Workers): це компоненти, які відповідають за виконання фактичних завдань (тобто обробку даних, запуск скриптів тощо). Робітники підключаються до сервера Airflow та отримують задачі для виконання. Вони можуть бути розташовані на різних серверах та мати різну конфігурацію, що дозволяє забезпечити гнучкість та масштабованість процесів обробки даних.

Ці три компоненти взаємодіють між собою, щоб забезпечити виконання процесів з обробки даних відповідно до заданого розкладу та конфігурації. Сервер Airflow керує роботою робітників та зберігає всі метадані у базі даних. Кожен робітник підключається до сервера Airflow та отримує завдання для виконання. У разі помилки або невдалого завершення завдання, Airflow Server сповіщає про це та може спробувати перезапустити завдання на іншому робітнику.

Ця архітектура дозволяє забезпечити гнучкість та масштабованість процесів обробки даних, а також забезпечує безпеку та надійність виконання завдань. Крім того, Apache Airflow підтримує різноманітні сервіси та інтеграції, що дозволяє використовувати його для різних завдань обробки даних, включаючи ETL-процеси, машинне навчання, обробку потокових даних тощо.

Ще одна важлива особливість архітектури Apache Airflow - це те, що вона

базується на концепції DAG (Directed Acyclic Graph). DAG є графічним представленням процесу обробки даних, в якому вузли відображають етапи обробки, а ребра вказують залежності між ними. У Airflow DAG використовується для визначення послідовності виконання завдань та їх залежностей.

1.3 Основні компоненти Apache Airflow

Ключові компоненти Apache Airflow дозволяють забезпечити ефективне та гнучке управління процесами обробки даних, що дає змогу використовувати цю технологію для автоматизації різноманітних задач. Ключові компоненти Apache Airflow включають:

1. Сервер Airflow: Це головний компонент, який управляє всіма іншими компонентами та процесами в Apache Airflow. Сервер Airflow містить веб-інтерфейс, де користувачі можуть переглядати та керувати всіма своїми процесами.
2. Двійкова база даних: Apache Airflow використовує базу даних для зберігання конфігураційних даних та інформації про стан виконання процесів. База даних може бути різних типів, таких як MySQL, PostgreSQL, SQLite тощо.
3. Scheduler: Цей компонент відповідає за планування та виконання задач відповідно до визначеної послідовності та залежностей, які визначені у DAG. Scheduler зчитує інформацію про DAG та запускає задачі на виконання.
4. Executor: Цей компонент відповідає за виконання задач, які запускає Scheduler. Executor може бути налаштований на використання різних підходів до виконання, таких як LocalExecutor, CeleryExecutor, KubernetesExecutor тощо.
5. Робітники (Workers): Це процеси, які відповідають за виконання конкретних задач, що були запущені Scheduler. Робітники можуть бути налаштовані на використання різних платформ та інфраструктури, таких як Docker, Kubernetes, AWS тощо.

На рисунку 1.1 зображено діаграму ключових компонентів Apache Airflow та залежностей між ними.

Рисунок 1.1 – Ключові компоненти Apache Airflow

1.4 Термінологія та базові концепції

В Apache Airflow є кілька базових концепцій та термінів, які корисно знати, щоб краще розуміти, як працює ця технологія. Основні терміни та концепції включають:

1. DAG (Directed Acyclic Graph) - це графічне представлення процесу, який складається зі списку задач та залежностей між ними. DAG визначає послідовність та залежності між задачами та показує, як вони пов'язані між собою.
2. Задача (Task) - це одна з операцій, яку необхідно виконати в процесі. Задачі виконуються послідовно та можуть мати залежності від інших задач, що визначено в DAG.
3. Оператор (Operator) - це конкретна реалізація задачі, яка виконує певну дію.

В Apache Airflow є багато вбудованих операторів, таких як BashOperator, PythonOperator, SQLOperator тощо, але можна створювати власні оператори з використанням API.

4. Виконавець (Executor) - це компонент, який відповідає за виконання задачі. У Airflow є кілька типів виконавців, таких як LocalExecutor, CeleryExecutor, KubernetesExecutor тощо.
5. Планувальник (Scheduler) - це компонент, який відповідає за планування та виконання задач відповідно до визначеної послідовності та залежностей, які визначені у DAG.
6. Веб-сервер (Web Server) - це інтерфейс користувача для Airflow, де можна переглядати, створювати та запускати DAG та задачі. Веб-сервер дає користувачам можливість керувати процесами та переглядати їх стан.
7. Підключення (Connection) - це налаштування, які дозволяють Airflow зв'язуватися з іншими системами та ресурсами. Підключення можуть містити інформацію про бази даних, API ключі, SSH ключі тощо.
8. Змінні (Variable) - це глобальні змінні, які можуть використовуватися в DAG та задачах. Змінні можуть містити будь-яку інформацію, таку як URL, параметри підключення, інформацію про те, чи була вже виконана якась операція тощо.
9. Hooks - це API, яке дозволяє з'єднуватися з зовнішніми ресурсами та виконувати дії з ними, такі як відправка електронних листів, взаємодія з AWS S3, взаємодія з базою даних тощо.
10. XCom - це механізм обміну даними між задачами в одному DAG. Задачі можуть передавати дані через XCom, щоб інші задачі могли використовувати ці дані для виконання своєї роботи.

1.5 Планування та тригери

Планувальник Airflow відстежує всі завдання і всі групи DAG і запускає екземпляри завдань, чиї залежності були виконані. За лаштунками він запускає

підпроцес, який відстежує і синхронізується з папкою для всіх об'єктів DAG, які вона може містити, а також періодично (приблизно щохвилини) збирає результати розбору DAG і перевіряє активні завдання на предмет можливості їх запуску.

Планувальник Airflow розроблений для запуску як постійна служба у виробничому середовищі Airflow. Щоб запустити його, все, що вам потрібно зробити, це запустити команду `airflow scheduler`. Він буде використовувати конфігурацію, вказану у файлі airflow.cfg.

Кожна DAG може мати або не мати розкладу, який інформує про те, як створюються запуски DAG. schedule_interval визначається як аргумент DAG, і отримує бажано вираз cron у вигляді str або об'єкт datetime.timedelta. Крім того, також можна використовувати один з параметрів, вказаний в таблиці 1.1.

Airflow DAG з датою початку (start_date), датою закінчення (end_date) та інтервалом (schedule_interval) визначає серію інтервалів, які планувальник перетворює на окремі запуски Dag Runs і виконує. Ключовою можливістю Airflow є те, що ці запуски DAG є атомарними, недієздатними елементами, і планувальник, за замовчуванням, перевіряє час життя DAG (від початку до кінця, по одному інтервалу за раз) і запускає DAG для будь-якого інтервалу, який не був запущений (або був очищений). Ця концепція називається наздоганяння (Catchup).

Таблиця 1.1 – Параметри розкладу

Параметр	Значення	cron-значення
None	Не використовує планування	
@once	Заплановує тільки один раз	
@hourly	Запускає кожну годину	0 * * * *
@daily	Запускає кожного день	0 0 * * *
@weekly	Запускає кожного тижня	0 0 * * 0
@monthly	Запускає кожного місяця	0 0 1 * *
@yearly	Запускає кожного року	0 0 1 1 *

1.6 Механізм обміну даними між задачами

XCom або Cross-Communication, це механізм Apache Airflow для передавання параметрів з одного оператора в інший. Якщо просто, то це таблиця в базі даних, що зберігає значення, записані операторами Airflow. У цій таблиці є кілька стовпців:

key - ключ запису

value - значення, дані зберігаються в двійковому форматі. Для MySQL це тип BLOB, а для Postgres BYTEA.

timestamp - час створення запису в базі

execution_date - дата виконання DAG

task_id - ID оператора, який записав дані в XCom

dag_id - ID DAG

Незважаючи на те, що BYTEA в Postgres може зберігати до 1 Гб двійкових даних, спільнота не рекомендує передавати через XCom великі дані. Це пов'язано з тим, що перед укладанням у таблицю вони піддаються серіалізації (JSON/pickle), а під час читання відбувається етап десеріалізації. Також не варто забувати, що дані передаються мережею, а це значно уповільнює роботу пайплайна.

Оператори можуть виконуватися в різних адресних просторах (Local

Executor) і навіть на окремих фізичних машинах (Celery/Task Executors, Kubernetes Executor). У такій ситуації потрібен механізм для передачі повідомлень від однієї машини до іншої. Центральним сховищем у цьому випадку є база даних Airflow (рекомендується використовувати PostgreSQL).

За замовчуванням якщо оператор у методі .execute повертає значення, це значення автоматично потрапляє в XCom. Цю поведінку можна змінити, передавши в аргумент `do_xcom_push` значення `False`:

```
from airflow.operators.bash import BashOperator

BashOperator(
   task_id='list_dir',
   bash_command='ls -la',
   do_xcom_push=False,
)
```

Якщо callable, переданий в аргументі `python_callable` у `PythonOperator`, повертає будь-яке значення, то це значення також автоматично буде записано в XCom. Приклад коду:

```
from airflow.operators.python import PythonOperator

def return_something():
   return 1 + 1

PythonOperator(task_id='return_something', python_callable=return_something)
```

Крім цього, з XCom можна працювати вручну, записуючи або отримуючи дані шляхом виклику методів `xcom_push` і `xcom_pull` у похідних від TaskInstance (до речі, теж таблиця в БД).

TaskInstance доступний у контексті виконання DAG, а також як змінна в шаблоні під ім'ям ti або task_instance. Наприклад, якщо вам необхідно прочитати значення попереднього таска всередині функції `PythonOperator`, то необхідно з контексту отримати TaskInstance і викликати метод `xcom_pull`, передавши йому назву оператора, який поклав результат у XCom:

```
def print_xcom(**kwargs):
   ti = kwargs['ti']
   print('The value is: {}'.format(ti.xcom_pull(task_ids='hello_world')))
```

```
with DAG(dag_id='xcom_dag', schedule_interval='@once') as dag:
   cmd = BashOperator(task_id='hello_world', bash_command='echo "Hello"')
   printer = PythonOperator(task_id='printer', python_callable=print_xcom)
   cmd >> printer
```

Усі змінні з контексту DagRun передаються в `python_callable` як іменовані аргументи. Також контекст всередині функції можна отримати шляхом виклику функції `get_current_context`, це зручно при використанні TaskFlow API:

```
from airflow.operators.python import get_current_context

@task
def download_file():
   context = get_current_context()
   return download_dataset(context['execution_date'].strftime('%Y-%m'))
```

На рисунку 1.2 зображено сторінку XCom, яка доступна через веб-інтерфейс Apache Airflow у розділі Admin → XComs.

Рисунок 1.2 – Приклад запуску DAG

Зверніть увагу на колонку key, у неї за замовчуванням записується значення return_value, що, є індикатором того, що оператор у методі execute повернув значення. Під час самостійного запису в XCom можна вказати свій key:

```
def push_cmd(ti):
   ti.xcom_push(value='cat /etc/passwd | wc -l', key='passwd_len')
def pull_result(ti):
   print(ti.xcom_pull(task_ids='bash_executor'))
```

2 ВСТАНОВЛЕННЯ ТА НАЛАШТУВАННЯ APACHE AIRFLOW

2.1 Встановлення та налаштування Apache Airflow

Найпростіший спосіб встановити Apache Airflow це використовувати docker-compose.yaml. У цьому файлі в розділі під назвою services ми визначаємо контейнери. Кожен контейнер запускає потрібний сервіс airflow, а саме: планувальник, веб-сервер та базу даних mysql.

Перш за все, давайте подивимось на `<<: *airflow-common`. Це перший крок, визначений у кожному сервісі. Знак `<<` називається якірним форматом YAML, який використовується для посилання на оголошення вмісту. Ми визначаємо цей вміст `airflow-common` зверху, починаючи з символу `&`. Цей синтаксис допомагає скоротити код yaml, повторно використовуючи вміст. Під `airflow-common` ми визначаємо конфігурації, які використовуються всіма сервісами.

```
version: '3'

x-airflow-common:
 &airflow-common
 image: apache/airflow:2.1.1-python3.8
 environment:
   &airflow-common-env
   AIRFLOW__CORE__SQL_ALCHEMY_CONN:
mysql://airflow:airflow@mysql:3306/airflow
   AIRFLOW__CORE__LOAD_EXAMPLES: 'true'
 volumes:
   - ./dags:/opt/airflow/dags
   - ./logs:/opt/airflow/logs
   - ./plugins:/opt/airflow/plugins
 user: "${AIRFLOW_UID:-50000}:${AIRFLOW_GID:-50000}"
 depends_on:
   mysql:
     condition: service_healthy
```

Контейнер mysql використовується для запуску MySQL бази даних у віртуальному середовищі. Це дозволяє легко встановлювати та керувати MySQL базою даних без необхідності вручну налаштовувати сервер та його залежності.

```
services:
```

```
mysql:
  image: mysql:5.7
  command: --explicit_defaults_for_timestamp
  ports:
    - "3306:3306"
  volumes:
    - my-db:/var/lib/mysql
```

Тут також визначені змінні оточення, такі як ім'я користувача, пароль, назва бази даних та інші. Додатково, контейнер має перевірку здоров'я (healthcheck), яка регулярно перевіряє доступність MySQL сервера в контейнері, щоб переконатися, що він працює належним чином.

```
  environment:
    MYSQL_ROOT_USER: airflow
    MYSQL_ROOT_PASSWORD: airflow
    MYSQL_USER: airflow
    MYSQL_PASSWORD: airflow
    MYSQL_DATABASE: airflow
  healthcheck:
    test: ["CMD", "mysqladmin" ,"ping", "-h", "localhost", '-pairflow']
    interval: 2s
    retries: 15
```

Контейнер `airflow-init` спочатку виконує всі кроки, згадані в `airflow-common`. Стандартна команда точки входу, що використовується в образі airflow, – `airflow`, яку ми замінимо користувацькими скриптами bash. Перші дві команди ініціалізують базу даних airflow: `airflow db init` та `airflow db upgrade`. І, нарешті, ми створимо користувача адміністратора за допомогою команди `airflow users create`. Це найкраще місце для додавання інших команд ініціалізації, таких як налаштування користувацьких з'єднань, секретів, додавання нової ролі тощо.

```
airflow-init:
  <<: *airflow-common
  entrypoint: /bin/bash -c "/bin/bash -c \"$${@}\""
  command: |
    /bin/bash -c "
      airflow db init
      airflow db upgrade
      airflow users create -r Admin -u admin -e airflow@airflow.com -f admin
-l user -p airflow
    "
```

```
  environment:
    <<: *airflow-common-env
```

Контейнер `airflow-scheduler` запускає основний компонент airflow, який називається планувальник. Ми просто запускаємо команду `scheduler`. Політика перезапуску контейнера встановлена на завжди, тому docker-compose подбає про перезапуск цього контейнера у разі будь-яких збоїв.

```
airflow-scheduler:
  <<: *airflow-common
  command: scheduler
  environment:
    <<: *airflow-common-env
  restart: always
```

Контейнер `airflow-webserver` буде запускати веб-сервер airflow за допомогою команди `webserver`. Ми також відкриваємо внутрішній порт контейнера 8080 для хост машини на 8081.

```
airflow-webserver:
  <<: *airflow-common
  command: webserver
  ports:
    - 8081:8080
  environment:
    <<: *airflow-common-env
```

Healthcheck дозволяє docker-compose визначити працездатність контейнера, виконавши відповідну команду в розділі test.

```
  healthcheck:
    test: ["CMD", "curl", "--fail", "http://localhost:8080/health"]
    interval: 10s
    timeout: 10s
    retries: 5
```

Користувачам Mac і Linux потрібно додатково експортувати деякі змінні середовища використовуючи наступну команду: `echo "AIRFLOW_UID=$(id -u)" > .env && echo "AIRFLOW_GID=0" >> .env`. Це створить файл .env, який буде завантажено у всі контейнери. Він подбає про те, щоб дозволи користувача були однаковими між томами каталогів хоста і каталогів контейнерів.

При першому запуску, потрібно запустити тільки контейнер `airflow-init`, який налаштує базу даних і створить користувача admin. Це можна зробити

запустивши наступну команду: `docker-compose up airflow-init`. Виконання цієї команди займе кілька хвилин. Після завершення, настав час запустити всі контейнери командою `docker-compose up`.

Якщо все пройшло успішно, отримати доступ до веб-сервера можна за адресою http://localhost:8081/. Використовуйте облікові дані користувача admin (створені на кроці `airflow-init`) для входу на веб-сервер. Після успішного входу відкриється головна сторінку інтерфейсу веб-сервера, яка зображена на рисунку 2.1.

Рисунок 2.1 – Головна сторінка Airflow

2.2 Створення DAG для автоматизації простих задач

DAG (Directed Acyclic Graph) - це складна робоча послідовність, що складається з завдань, які виконуються в певному порядку залежно від їх взаємозв'язків. У Apache Airflow DAG є об'єктом, який представляє таку послідовність, і він може бути використаний для автоматизації різних задач.

Створення DAG в Apache Airflow відбувається шляхом створення Python-файлу, що містить код, який визначає залежності між завданнями та їх параметри.

Ось приклад простого DAG для запуску скрипту Python. Першим кроком підключаємо всі потрібні бібліотеки:

```
from datetime import datetime, timedelta
from airflow import DAG
from airflow.operators.bash_operator import BashOperator
```

Потім задаємо налаштування за замовчуванням:

```
default_args = {
   'owner': 'airflow',
   'depends_on_past': False,
   'start_date': datetime(2023, 4, 4),
   'email_on_failure': False,
   'email_on_retry': False,
   'retries': 1,
   'retry_delay': timedelta(minutes=1),
}
```

Створюємо DAG. Задаємо назву, опис, інтервал виконання та попередньо створені налаштування:

```
dag = DAG(
   'my_dag',
   default_args=default_args,
   description='Run a Python script',
   schedule_interval=timedelta(days=1),
)
```

Останнім кроком створюємо bash оператор:

```
run_script = BashOperator(
   task_id='run_script',
   bash_command='python /path/to/your/script.py',
   dag=dag,
```

```
)

run_script
```

Цей DAG виконується щодня, запускає скрипт Python та надсилає повідомлення електронної пошти у разі помилки виконання. Щоб скористатися цим DAG, необхідно зберегти його як Python-файл у директорії ./dags (або в будь-якій іншій визначеній директорії) та запустити процес Airflow.

На рисунку 2.2 зображено приклад історії виконань DAG, включаючи успішні, невдалі, в черзі на виконання, пропущені та інші стани.

Рисунок 2.2 – Історія виконань DAG

Також, слід пам'ятати, що DAG може містити багато завдань та параметрів, що дозволяє створювати складніші послідовності для автоматизації ваших завдань.

Також важливим аспектом створення DAG є визначення залежностей між завданнями. Залежності визначаються за допомогою операторів в Apache Airflow.

Наприклад, щоб додати залежність між двома завданнями, слід визначити ці завдання та встановити одне з них як "батьківське", а інше - як "дочірнє". Це можна зробити за допомогою параметрів upstream та downstream операторів. Ось приклад створення DAG з залежністю між завданнями. Створюємо батьківський

оператор:

```
run_script1 = BashOperator(
   task_id='run_script1',
   bash_command='echo "Hello World!"',
   dag=dag,
)
```

Створюємо дочірній оператор:

```
run_script2 = BashOperator(
   task_id='run_script2',
   bash_command='echo "Hello Again!"',
   dag=dag,
)
```

Визначаємо залежність між ними:

```
run_script1 >> run_script2
```

У цьому прикладі, `run_script1` є батьківським завданням, а `run_script2` є дочірнім завданням. Параметр `>>` встановлює залежність між завданнями. Це означає, що `run_script2` не буде запущено, поки не завершиться виконання `run_script1`.

Це базовий опис створення DAG з залежністю між завданнями в Apache Airflow. Використовуючи цей підхід, можна створювати більш складні послідовності для автоматизації ваших завдань з використанням багатьох різних операторів.

2.3 Використання операторів Apache Airflow

Apache Airflow містить багато різноманітних операторів, що дозволяють виконувати різні завдання в рамках DAG. Оператори можуть бути використані для виконання різних дій, таких як виконання скриптів Python, Bash, SQL запитів до бази даних, надсилання електронної пошти, завантаження файлів та багато іншого. Ось деякі з найбільш поширених операторів в Apache Airflow:

1. BashOperator – дозволяє виконувати Bash скрипти на виконання різних завдань.
2. PythonOperator – дозволяє виконувати Python скрипти для виконання різних

завдань.

3. EmailOperator – дозволяє надсилати електронну пошту з використанням SMTP-сервера.
4. SimpleHttpOperator – дозволяє виконувати HTTP-запити до API.
5. MySqlOperator / PostgresOperator – дозволяє виконувати SQL-запити до бази даних.
6. PythonVirtualenvOperator – дозволяє виконувати Python скрипти у віртуальному середовищі Python.

Ці оператори можна поєднувати між собою, щоб створювати більш складні послідовності дій для виконання завдань в рамках DAG. Наприклад, можна використовувати PythonOperator для запуску скрипта Python, який отримує дані з бази даних за допомогою MySqlOperator, обробляє ці дані та зберігає їх у файлі за допомогою BashOperator.

Також можна створити свій власний оператор, який відповідатиме за виконання спеціалізованого завдання. Для цього потрібно створити підклас `BaseOperator` і перевизначити методи `execute` та `template_fields`.

```
class MyCustomOperator(BaseOperator):
   @apply_defaults
   def __init__(self, my_param, *args, **kwargs):
       super(MyCustomOperator, self).__init__(*args, **kwargs)
       self.my_param = my_param

   def execute(self, context):
       pass # Виконання завдання

   def template_fields(self):
       return ('my_param',)
```

За допомогою свого власного оператора, ви можете виконувати складні завдання, які не можуть бути виконані з використанням стандартних операторів. Використання правильного оператора для кожного етапу вашого DAG допоможе підвищити ефективність та стабільність вашого пайплайну.

2.4 Використання сенсорів Apache Airflow

Сенсор - це тип оператора, який перевіряє, чи виконується умова через певний проміжок часу. Якщо умова виконана, завдання позначається як успішне, і група DAG може переходити до наступних завдань. Якщо умова не виконана, датчик чекає на наступний інтервал, перш ніж перевіряти знову.

Усі датчики успадковуються від BaseSensorOperator і мають такі параметри:

- mode: Режим сенсора:
  - poke: Це режим за замовчуванням. При використанні poke датчик займає робочий слот на весь час виконання і спить між тиканнями. Цей режим найкраще використовувати, якщо очікується короткий час роботи датчика.
  - reschedule: У цьому режимі, якщо критерії не виконано, датчик звільняє свій робочий слот і переносить наступну перевірку на більш пізній час. Цей режим найкраще використовувати, якщо ви очікуєте тривалий час роботи датчика, оскільки він менш ресурсоємний і звільняє робітників для інших завдань.
- poke_interval: це час у секундах, який датчик чекає перед повторною перевіркою стану. За замовчуванням - 60 секунд. Використовується тільки при використанні режиму poke.
- exponential_backoff: Якщо встановлено у True, цей параметр створює експоненціально довший час очікування між тиканнями у режимі тикання.
- timeout: Максимальний проміжок часу у секундах, протягом якого датчик перевіряє умову. Якщо умова не буде виконана протягом зазначеного періоду, завдання не буде виконано.
- soft_fail: Якщо встановлено True, завдання позначається як пропущене, якщо умова не виконана протягом таймауту.

Багато пакетів провайдерів Airflow містять датчики, які очікують на різні критерії в різних системах-джерелах. Нижче наведено деякі з найпоширеніших датчиків:

- @task.sensor: Дозволяє перетворити будь-яку функцію Python, яка повертає значення PokeReturnValue, на екземпляр класу BaseSensorOperator. Цей спосіб створення сенсора корисний при перевірці складної логіки або якщо ви підключаєтеся до інструменту через API, який не має спеціального сенсора.
- S3KeySensor: Очікує на появу ключа (файлу) у відрі Amazon S3. Цей датчик корисний, якщо ви хочете, щоб ваша група DAG обробляла файли з Amazon S3 в міру їх надходження.
- DateTimeSensor: Очікує на вказану дату і час. Цей датчик корисний, якщо ви хочете, щоб різні завдання в одній групі DAG виконувалися в різний час.
- ExternalTaskSensor: Очікує на завершення завдання Airflow. Цей датчик корисний, якщо ви хочете реалізувати крос-групові залежності в одному середовищі Airflow.
- HttpSensor: Очікує на доступність API. Цей датчик корисний, якщо ви хочете переконатися, що ваші запити до API є успішними.
- SqlSensor: Очікує на наявність даних у таблиці SQL. Цей датчик корисний, якщо ви хочете, щоб група DAG обробляла дані в міру їх надходження до бази даних.

Використовуючи сенсори, щоб уникнути потенційних проблем з продуктивністю, потрібно пам'ятати про наступне:

- Завжди визначайте значущий параметр таймауту для вашого датчика. За замовчуванням цей параметр дорівнює семи дням, що є задовгим часом для роботи вашого датчика. При використанні сенсора, потрібно подумати про сценарій використання і про те, як довго ви очікуєте, що датчик буде чекати, а потім точно визначити цього таймаут.
- Коли це можливо, особливо для довготривалих датчиків, використовуйте режим перепланування, щоб ваш датчик не був постійно зайнятий робочим слотом. Це допоможе уникнути тупикових ситуацій у Airflow, коли датчики займають усі доступні робочі слоти.
- Якщо ваш poke_interval дуже короткий (менше 5 хвилин), використовуйте

режим тикання. Використання режиму перепланування в цьому випадку може перевантажити ваш планувальник.

- Визначте значущий poke_interval на основі вашого сценарію використання. Немає необхідності перевіряти умову кожні 60 секунд (за замовчуванням), якщо ви знаєте, що загальний час очікування складе 30 хвилин.

Починаючи з Airflow версії 2.5, можна використовувати декоратор `@task.sensor` з API TaskFlow для використання будь-якої функції Python, яка повертає PokeReturnValue як екземпляр BaseSensorOperator.

Наступний приклад коду DAG показує, як використовувати декоратор `@task.sensor`. Для початку, підключаємо потрібні бібліотеки:

```
from airflow.decorators import dag, task
from airflow.sensors.base import PokeReturnValue
from pendulum import datetime
import requests
```

Оголошуємо DAG декоратор, який встановлює параметри графа обробки задач. У нашому випадку, ми використовуємо декоратор @dag для встановлення дати початку (start_date), розкладу (schedule), та параметра catchup. Значення start_date встановлено на 1 грудня 2022 року, розклад виконання задач є щоденним (@daily), а catchup=False вказує, що необхідно пропустити пропущені виконання задач при старті DAG.

```
@dag(start_date=datetime(2022, 12, 1), schedule="@daily", catchup=False)
def sensor_decorator():
```

Оголошуємо функцію check_shibe_availability, яка буде виконувати перевірку доступності URL. Ця функція використовує декоратор `@task.sensor` для вказівки, що це сенсорний оператор, який перевіряє стан задачі з певною періодичністю. Параметри сенсора встановлюються через аргументи декоратора, такі як poke_interval, timeout і mode. У нашому випадку, poke_interval=30 вказує, що сенсор буде перевіряти стан задачі кожні 30 секунд, timeout=3600 означає, що максимальний час очікування для сенсора становить 3600 секунд (1 година), і mode="poke" вказує, що сенсор буде виконувати перевірку шляхом "вторгання" (poke).

```
    @task.sensor(poke_interval=30, timeout=3600, mode="poke")
    def check_shibe_availability() -> PokeReturnValue:
        r = requests.get("http://shibe.online/api/shibes?count=1&urls=true")
        if r.status_code == 200:
            condition_met = True
            operator_return_value = r.json()
        else:
            condition_met = False
            operator_return_value = None

        return PokeReturnValue(is_done=condition_met,
xcom_value=operator_return_value)
```

Додамо ще один оператор `print_shibe_picture_url` до DAG. Цей оператор буде викликати функцію `print_shibe_picture_url` з аргументом `url`, який отримується з результату сенсорного оператора `check_shibe_availability`().

```
    @task
    def print_shibe_picture_url(url):
        print(url)

    print_shibe_picture_url(check_shibe_availability())

sensor_decorator()
```

2.5 Використання Telegram хуків та відсилання повідомлень

Apache Airflow має можливість надсилання повідомлень на основі хуків (hooks). Хук (hook) - це інтерфейс, що дозволяє взаємодіяти з різними зовнішніми системами.

Один з найпопулярніших способів отримання повідомлень - це використання Telegram. Для відправки повідомлень на Telegram потрібно створити Telegram бота та отримати API token. Після цього можна використовувати Telegram API для надсилання повідомлень на ваш Telegram обліковий запис.

Щоб налаштувати Telegram хук, потрібно додати наступний код до файлу airflow.cfg:

```
[telegram]
```

```
token = <API_TOKEN>
chat_id = <CHAT_ID>
```

де `<API_TOKEN>` - API token Telegram бота, `<CHAT_ID>` - ідентифікатор чату, до якого будуть надсилатись повідомлення.

Після цього можна використовувати `TelegramHook`, наприклад, для надсилання повідомлення про завершення DAG.

Для початку, імпортуємо потрібні бібліотеки. Слід пам’ятати, що для використання `TelegramHook` необхідно встановити додатковий пакет pip `apache-airflow-providers-telegram`.

```
from airflow.providers.telegram.hooks.telegram import TelegramHook
from airflow.operators.python_operator import PythonOperator
from datetime import datetime
```

Створюємо функцію для відправки повідомлення в месенджер Telegram. Спочатку, створюємо об'єкт `TelegramHook`, який ініціалізується з параметром `telegram_conn_id='telegram'`. `TelegramHook` є об'єктом з'єднання, який надає методи для взаємодії з Telegram API.

Після створення об'єкту `TelegramHook` функція формує повідомлення, використовуючи поточний час в форматі "`{datetime.now()}`". Це означає, що в повідомленні буде відображатись час виконання функції.

Далі, за допомогою методу `send_message`(), надсилаємо повідомлення в Telegram. Параметр `chat_id` вказує ідентифікатор чату, до якого слід надіслати повідомлення. Потрібно замінити `CHAT_ID` на фактичний ідентифікатор чату, до якого ви хочете надіслати повідомлення. Параметр text містить саме повідомлення, яке буде надіслано.

```
def send_telegram_message():
   hook = TelegramHook(telegram_conn_id='telegram')
   message = f"DAG execution finished at {datetime.now()}"
   hook.send_message({chat_id: 'CHAT_ID', text: message})
```

Далі використовуємо цю функцію в Python операторі:

```
telegram_task = PythonOperator(
   task_id='telegram_task',
   python_callable=send_telegram_message,
   dag=dag
```

)

3 ПРАКТИЧНА РЕАЛІЗАЦІЯ СИСТЕМИ АВТОМАТИЗАЦІЇ ЗАДАЧ З ВИКОРИСТАННЯМ APACHE AIRFLOW

3.1 Створення DAG для автоматичного надсилання сповіщень про повітряну тривогу

Створення DAG в Apache Airflow відбувається шляхом створення Python-файлу, що містить код, який визначає залежності між завданнями та їх параметри. Цей файл повинен знаходитись у папці dags. DAG (Directed Acyclic Graph) є концептуальним поняттям, яке використовується для визначення та організації пайплайнів даних або робочих процесів. DAG визначає послідовність тасків та залежності між ними. Кожен таск представляє певну операцію або крок в пайплайні. Таски виконуються в певному порядку відповідно до визначених залежностей.

Загальний вигляд кроків розроблюваного DAG зображено на рисунку 3.1:

Рисунок 3.1 – Блок-схема залежностей між задачами

Для початку імпортуємо потрібні бібліотеки:

1. pendulum: Ця бібліотека надає розширений функціонал для роботи з датами і часом, який перевершує стандартні засоби Python. Вона використовується для роботи з датами та часом в цьому коді.
2. time: Ця бібліотека надає функції для роботи з часом в мілісекундах. Вона використовується для певних операцій з часом.

3. requests: Ця бібліотека дозволяє здійснювати HTTP-запити (GET, POST, PUT, DELETE і т.д.) до веб-серверів. Вона використовується для звернення до зовнішнього веб-серверу для отримання або надсилання даних.
4. json: Ця бібліотека дозволяє працювати з форматом JSON (JavaScript Object Notation), який широко використовується для обміну даними. Вона використовується для перетворення даних в формат JSON або зчитування даних з JSON-рядка.
5. airflow: Це фреймворк для планування та управління робочими процесами (workflow) великого масштабу. Він надає інструменти для організації та автоматизації складних процесів обробки даних. У цьому коді використовується для створення та запуску DAG (Directed Acyclic Graph), який є графічним представленням послідовності задач.
6. airflow.exceptions: Ця бібліотека містить винятки, які використовуються для обробки виключень, пов'язаних з плануванням та виконанням робочих процесів в Airflow.
7. airflow.operators.python_operator: Цей модуль надає оператори, які дозволяють виконувати функції Python у визначеній послідовності в рамках робочого процесу

```
import pendulum
import time
import requests
import json

from airflow import DAG
from airflow.exceptions import AirflowSkipException
from airflow.operators.python_operator import PythonOperator
from airflow.providers.telegram.hooks.telegram import TelegramHook
from datetime import datetime, timedelta
```

Наступним кроком напишемо оператор Apache Airflow (у вигляді функції), який буде перевіряти статус тривоги. Після досліджень різний сервісів, було вибрано ubilling.net.ua. За допомогою їх API можна легко отримання інформацію про тривогу у всіх областях України.

Змінна `__schedule_interval` задає інтервал між перевірками статусу тривоги. Було обрано інтервал у 2 хвилини.

```
__schedule_interval=timedelta(minutes=2)
```

Функція `check_air_alerts(ti)` викликає функцію `__query_air_alerts()`, яка в свою чергу викликає API, перевіряє на наявність змін, та повертає області, в який нещодавно почалася чи закінчилась тривога. Після цього, функція `check_air_alerts(ti)` зберігає результат у XCom.

```
def check_air_alerts(ti):
   states = list(__query_air_alerts())
   states_json = json.dumps(states)

   ti.xcom_push(key='state', value=states_json)
```

XCom (Cross Communication) є функціональним аспектом Apache Airflow, який дозволяє обмінюватись даними (результатами обчислень, станами тощо) між тасками, що виконуються в рамках пайплайну даних в Apache Airflow. XCom реалізується за допомогою спеціальної бази даних, яка зберігає дані XCom після їх записування в одній тасці і передачі до іншої. Ця база даних може бути реалізована через реляційну базу даних, таку як PostgreSQL або MySQL, або через інші механізми зберігання даних, такі як Redis або S3. Для запису та отримання даних використовуються методи `ti.xcom_push()` та `ti.xcom_pull()` відповідно.

```
def __query_air_alerts():
   response = requests.get('https://ubilling.net.ua/aerialalerts/')
   response.raise_for_status()
   data = response.json()
   current_time = datetime.now() + timedelta(hours=3)

   for state_name, state in data['states'].items():
       changed_at = datetime.strptime(state['changed'], '%Y-%m-&d %H:%M:%S')
       if current_time - changed_at > __schedule_interval:
           continue
       yield {'state_name': state_name, 'is_on': state['alertnow']}
```

Наступним кроком створюємо функцію `send_notification(ti)`, яка буде відправляти повідомлення про початок та відбій тривоги у Telegram чат. Для цього використовуємо телеграм хук з пакету `airflow.providers.telegram.hooks.telegram`.

```
def send_notification(ti):
   states_json = ti.xcom_pull(key='state', task_ids='check_air_alerts')
   states = json.loads(states_json)

   format_func = lambda state : f"{state['state_name']} - {'повітряна
тривога' if state['is_on'] else 'відбій тривоги'}"

   message = '\n'.join(map(format_func, states))
   if message == '':
       raise AirflowSkipException
  message = f'!! Увага! !!\n\n{message}'

   telegram_hook = TelegramHook(telegram_conn_id='telegram_conn_id',
chat_id='0123456789')
   telegram_hook.send_message({'text': message})
```

Після створення потрібних функцій, прийшов час налаштування самого DAG. Задаємо його назву та інтервал його виконання.

```
with DAG(
   'telegram_alert',
   start_date=pendulum.datetime(2022, 1, 1, tz='UTC'),
   schedule_interval=__schedule_interval,
   catchup=False,
) as dag:
```

Також потрібно налаштувати токен від телеграм бота. Це можна зробити у файлі airflow.cfg або за допомогою environment variable під назвою `AIRFLOW_CONN_TELEGRAM_CONN_ID` та вмістом у форматі `{"password": "********"}`.

Створюємо оператор Python, який викликає попередньо створену функцію перевірки повітряної тривоги:

```
   check_air_alerts_operator = PythonOperator(
       task_id='check_air_alerts',
       python_callable=check_air_alerts,
       dag=dag
   )
```

Створюємо оператор Python, який викликає попередньо створену функцію відправлення сповіщення при повітряній тривозі:

```
   send_notification_operator = PythonOperator(
       task_id='send_notification',
       python_callable=send_notification,
       dag=dag,
```

```
    trigger_rule='all_success'
)
```

Задаємо правило запуску як `'all_success'`. Це потрібно, щоб оператор `send_notification_operator` запускався тільки у випадку успішного виконання всіх попередніх тасків у DAG.

За замовчуванням, якщо не задано жодного правила (`trigger_rule`), то оператор буде запускатись незалежно від статусу попередніх тасків. Однак, задавши правило `all_success`, ми вказуємо, що оператор `send_notification_operator` повинен виконуватись лише тоді, коли всі попередні таски успішно виконались (тобто не мають статусу `failed`, `skipped`, `upstream_failed`).

Це корисно, коли оператор `send_notification_operator` відповідає за надсилання сповіщення або повідомлення про завершення пайплайну або важливої фази робочого процесу. Задавши правило `all_success`, ми гарантуємо, що сповіщення буде надіслано тільки у разі успішного виконання всіх попередніх кроків або завершення DAG без помилок.

Останнім кроком встановлюємо залежність між задачами:

```
check_air_alerts_operator >> send_notification_operator
```

Цей крок встановлює залежність між задачами `check_air_alerts_operator` та `send_notification_operator`. Запис `>>` вказує, що `check_air_alerts_operator` повинен виконуватись перед `send_notification_operator`.

Це означає, що перед надсиланням повідомлення (`send_notification_operator`), спочатку буде виконано перевірку наявності повітряних сповіщень (`check_air_alerts_operator`). Це забезпечує послідовність виконання тасків у пайплайні. Графік залежностей зображено на рисунку 3.2.

Залежності між задачами визначають порядок їх виконання і дозволяють контролювати логіку пайплайну. В даному випадку, перед надсиланням сповіщення, ми перевіряємо наявність повітряних сповіщень, щоб впевнитись, що повідомлення надсилається тільки в разі їх наявності.

Рисунок 3.2 – Графік залежностей між задачами

3.2 Аналіз виконання створеного DAG

Щоб створений пайплайн запрацював, його слід активувати використовуючи веб інтерфейс. Це можна зробити на сторінці, яка відповідає за управління DAG-ами. Щоб увімкнути DAG, потрібно клацнути на перемикач біля його назви. На рисунку 3.3 показано, як цей перемикач повинен виглядати, коли він увімкнений.

Рисунок 3.3 – Перемикач ввімкнення та вимкнення DAG

На рисунку 3.4 зображено статистику його нещодавніх запусків. Кольори статусів мають наступні значення:

1. Зелений: Зелений колір означає, що DAG або задача були успішно виконані без помилок. Якщо стан DAG або задачі зелений, це свідчить про те, що вони завершилися успішно.
2. Рожевий: Рожевий колір вказує на те, що DAG або задача були виконані, але мали попередження або інші несерйозні помилки. Такий статус може вказувати на те, що в процесі виконання виникли деякі незначні проблеми, але вони не перешкоджають завершенню задачі. У випадку з оператором `send_notification` це свідчить про те, що не було змін до повітряної тривоги, тому сповіщення не було відправлено.
3. Червоний: Червоний колір показує, що DAG або задача мали серйозні помилки або зупинились через критичні проблеми. Якщо стан DAG або задачі червоний, це означає, що виникли серйозні проблеми, які потребують уваги та виправлення.

4. Жовтий: Жовтий колір вказує, що виконавча послідовність була перервана через помилки в їхніх попередниках. Це означає, що одне або декілька завдань, які мають бути виконані перед поточним DAG або задачею, не вдалося успішно завершити. Це може бути викликано різними причинами, такими як помилки в коді, недоступність ресурсів або проблеми з мережею.

Рисунок 3.4 – Історія виконань DAG

Щоб переглянути логи, потрібно нажати на квадратик навпроти оператора та відкривати вкладку за логами. Приклад логів оператора `send_notification` зображений на рисунку 3.5.

Рисунок 3.5 – Історія виконань DAG

Ось інформація, яка логується:

1. Перший рядок "3b826b225aa8" є ідентифікатором хоста або працівника, на якому виконується завдання. Це може бути корисно для відстеження місця виконання завдання в розподіленому середовищі.
2. Рядок "*** Found local files:" вказує на те, що було знайдено локальні файли, пов'язані із завданням.
3. У наступному рядку вказано шлях до певного файлу журналу, пов'язаного із завданням, із зазначенням місця, де можна знайти журнал для цієї спроби виконання завдання.
4. Наступні рядки містять детальну інформацію про виконання завдання, включно з позначками часу, рівнями журналу і конкретними подіями. Деяка інформація з журналу включає:
    a. Перевірка залежностей для завдання із зазначенням того, чи виконано ці залежності.
    b. Інформація про спроби запуску із зазначенням номера спроби виконання завдання.
    c. Деталі виконання завдання, такі як тип завдання (у цьому випадку

PythonOperator) і мітка часу виконання.

d. Інформація про процеси та команди, пов'язані з виконанням завдання.
e. Експорт змінних оточення, пов'язаних з контекстом виконання завдання.
f. Попередження або інформаційні повідомлення, такі як попередження про порожній криптографічний ключ.
g. Деталі HTTP-запиту, що вказують на вихідний API-запит до Telegram API в даному випадку.
h. Повернуте значення завдання (в даному випадку - None).
i. Оновлення статусу завдання, позначення завдання як УСПІШНОГО.
j. Статус завершення завдання та подальша інформація про завдання.

На рисунку 3.6 зображено таблицю з тривалістю виконання завдань, яка знаходиться на сторінці “Task Duration”. Всі завдання виконувались менше однієї секунди.

Рисунок 3.6 – Історія виконань DAG

На рисунку 3.7 зображено повідомлення, які відправляються ботом через Telegram hook.

Рисунок 3.7 – Сповіщення в Telegram

На рисунку 3.8 зображено повідомлення, які містять одразу декілька

областей.

Рисунок 3.8 – Сповіщення в Telegram з одразу декількома областями в одному повідомлення

Схожість

Джерела з Бібліотеки 8

1	**Студентська робота**	ID **файлу:** 1015015061	**Навчальний заклад:** Lutsk National Technical University	0.17%
2	**Студентська робота**	ID **файлу:** 1015200159	**Навчальний заклад:** Lviv Polytechnic National University	0.15%
3	**Студентська робота**	ID **файлу:** 1009680155	**Навчальний заклад:** Lviv Polytechnic National University	0.15%
4	**Студентська робота**	ID **файлу:** 1015146509	**Навчальний заклад:** Vinnytsia State Pedagogical Universit... 3 Джерело	0.13%
5	**Студентська робота**	ID **файлу:** 1008382240	**Навчальний заклад:** National Technical University of Ukraine "Kyi...	0.13%
6	**Студентська робота**	ID **файлу:** 1006144112	**Навчальний заклад:** National Technical University of Ukraine "Kyi...	0.13%