Chapter 4 : Airflow 콘텍스트를 사용하여 태스크 템플릿 작업하기

#증분 데이터 적재방법

[1] 데이터 확인

$ wget http://dumps.wikimedia.org/other/pageviews/2019/2019-07/pageviews-20190701-010000.gz

- **wget** : 웹 서버에서 파일 다운시 사용하는 명령어(Linux)
- **pageviews-20190701-010000** : 파일의 날짜/시간
  (ex) 210000는 20시00분~21시00분까지
- **url/{year}/{year-month}/pageviews-{year}{month}{day}{hour}0000.gz**

$ gunzip pageviews-20190701-010000.gz

- **gunzip** : 파일 압축 해제 명령어

$ head pageviews-20190701-010000

(결과) aa Main_Page 1 0

aa Community_page 1 0 .....

(1) (2) (3)(4)

- **head** : 파일의 앞부분 확인하는 명령어
- (1) 도메이 코드 / (2) 페이지 제목 / (3) 조회수 / (4) 응답크기(byte)

[2] 데이터 분석 (참고: 34p)

$ wget http://dumps.wikimedia.org/other/pageviews/2019/2019-07/pageviews-20190701-010000.gz

$ gunzip pageviews-20190701-010000.gz

$ awk -F' ' '{print $1}' pageviews-20190701-010000

- awk : 데이터 값 조작(핸들링)하는 데 필요한 명령어

#오퍼레이터의 인수 템플릿 작업

- 사용할 오퍼레이터 : **BashOperator**, **PythonOperator**

[1] BashOperator 사용

- " " (이중괄호) 의미 : 런타임 시 삽입될 변수 나타냄 (**Jinja 템플릿 문자열**)
  사용자가 런타임 시 값 입력함 -> 미리 값 알수 없음.

```
import airflow.utils.dates
from airflow import DAG
from airflow.operators.bash import BashOperator
dag=DAG(
        start_date=airflow.utils.dates.days_ago(3),
        schedule_interval="@hourly",
)
get_data=BashOperator(
        task_id="get_data",
        bash_command=(
                "curl -o /tmp/wikipageviews.gz"
                "https://dumps.wikimedia.org/other/pageviews/"
                "{{ execution_date.year }}/"
                "{{ execution.date.year}}-"
                "{{ '{:02}'.format(execution_date.month) }}/"
                "pageviews-{{ execution_date.year }}"
                "{{ '{:02}'.format(execution_date.month) }}"
                "{{ '{:02}'.format(execution_date.day) }}-"
                "{{ '{:02}'.format(execution_date.hour) }}0000.gz"
        ),
        dag=dag,
```

)

- **bash_command** 인수 사용(Bash 명령을 제공하는 인수임)
- {{ '**{:02}**'.format(execution_date.month) }}
  - ➔ Jinja 템플릿 문자열 내 "**패딩 문자열 형식**" 적용 (빈 앞자리 0 채움)
- **execution_date** 변수 사용
  - ➔ execution_date는 Pendulum의 datetime 객체 (Python datetime 호환 객체)
  - ➔ Airflow는 날짜/시간에 Pendulum 라이브러리 사용함
  - ➔ 파이썬에서 사용가능한 메서드를 Pendulum에도 사용 가능

(ex) >>> from datetime import datetime

>>> import pendulum

>>> datetime.now().year

2020 // 동일결과

>>> pendulum.now().year

2020 // 동일결과

[참고] task context (태스크 콘텍스트) 의미

: 작업(task)이 실행될 때 해당 작업에 제공되는 다양한 **메타데이터**와 **환경 변수**들을 의미

- **작업이 실행되는 동안** 유용한 정보를 제공하며, 작업 내에서 동적으로 활용 가능
- 컨텍스트 변수를 사용하면 작업 실행 중에 필요한 데이터를 동적으로 설정하고 처리할 수 있어 유연한 작업 흐름을 구축할 수 있음

(ex) 주요 task context의 항목

- ds, ds_nodash: 작업이 실행되는 날짜 (YYYY-MM-DD 형식)와 하이픈이 제거된 날짜.
- prev_ds, next_ds: 이전 실행 날짜 및 다음 실행 날짜.
- execution_date: 현재 작업의 실행 날짜 및 시간.
- dag: 현재 실행 중인 DAG 객체.
- task: 현재 실행 중인 Task 객체.
- ti (TaskInstance): 현재 작업의 TaskInstance 객체로, 이 객체를 통해 다양한 작업 상태 및 메타데이터에 접근할 수 있음.

[참고] {{ 인수 }} Jinja 템플릿에 적용 가능한 인수

- {{name}} 문자열은 Jinja 템플릿 가능한 "**속성 리스트**"에 포함되지 않으면 문자열 그대로 해석됨

➔ 이 리스트는 모든 Operator의 "**template_fields**" 속성에 의해 설정됨.

➔ template_fields 요소 = 클래스 속성 이름.

➔ 인수이름 = 클래스 속성 이름
(따라서, 인수가 매핑되는 클래스 속성에 대해 문서화 권면)

#PythonOperator 사용해 전체 "태스크 콘텍스트" 출력

```
import airflow.utils.dates
from airflow import DAG
from airflow.operators.python import PythonOperator
dag=DAG(
        dag_id="chapter4_print_context"
        start_date=airflow.utils.dates.days_ago(3),
        schedule_interval="@daily",
)
def _print_context(**kwargs):
        print(kwargs)
print_context=PythonOperator(
        task_id="print_context",
        python_callable=_print_context,
        dag=dag,
)
```

(결과)

```
{
        'dag': <DAG: print_context>,
        'ds': '2019-07-04'
        'next_ds': '2019-07-04'
        'next_ds_nodash': '20190704'
```

'prev_ds': '2019-07-03'

'prev_ds_nodash': '20190703'

}

- **python_callable** 인수 사용
- "태스크 콘텍스트"에서 사용가능한 모든 변수 리스트 출력
- 모든 변수 ****kwargs**에 저장되어 print() 함수에 전달
  (위 변수들은 런타임 시 사용 가능함)

[비교] BashOperator vs PythonOperator

BashOperator 사용 로직	PythonOperator 사용 로직
bash_command 인수 사용	Python_callable 인수 사용
> 런타임 시 자동으로 템플릿 지정/문자열 제공	> 템플릿 화 (x) / 콜러블(callable) 제공
> Bash명령을 문자열로 받아 실행	> 파이썬 함수를 이용해 실행

[참고] 콜러블(callable)

: 파이썬에서 호출할 수 있는 객체를 의미

- PythonOperator는 특정 파이썬 함수를 작업으로 실행할 때 사용되며, 이때 **함수** 자체가 "**콜러블**"로 전달됨.

```
// 간단한 파이썬 함수 정의
def my_function():
    print("Hello from Airflow!")
// PythonOperator에 콜러블 함수 전달
    task = PythonOperator(
        task_id='my_task',
        python_callable=my_function
    )
```

- 위 코드에서 my_function은 "콜러블"이며, 이 함수는 PythonOperator에 전달되어 Airflow 작업으로 실행
- Airflow는 이 함수를 실행할 때 필요한 인수를 전달할 수 있으며, 함수의 결과를 다른

작업으로 전달하거나 작업의 성공/실패 여부를 결정할 수 있음

- 요약하면, Airflow에서의 "콜러블"은 작업으로 실행할 수 있는 함수나 객체를 의미하며, 주로 PythonOperator와 같이 사용

#키워드 인수

: ****인수명** (ex: **context) : "단일 인수"에 "모든 키워드 인수" 포함시킬 수 있음

def _print_context**(**context**):

print(context)

[추가내용]

def _get_data(**execution_date, **context**):

- **기본값**으로 인수 설정 가능(execution_date)
- execution_date 인수 **外** 모든 인수들은 **context에 전달됨

#PythonOperator 변수 제공

- PythonOperator는 콜러블 함수에서 추가 인수 제공함
- **output_path**를 입력 가능하게 만들어, 작업에 따라 "출력 경로" 변경 가능 위해 전체 함수 복사 대신 output_path만 별도로 구성 가능

def _get_data(**output**, **context):

로직(~~)

request.urlretrieve(url, **output_path**)

#PythonOperator 콜러블 커스텀 변수 제공

(1) op_args 인수

- op_args 인수에 제공된 리스트의 각 값 "콜러블 함수"에 저달됨

- _get_data("/tmp/wikipageviews.gz") 함수 직접 호출 결과와 동일

(ex-1) def _get_data(**output_path**, **context):

함수 로직(~~)

get_data=**PythonOperator**(

task_id="get_data",

python_callable=_get_data,

op_args=["/tmp/wikipageviews.gz"],

dag=dag

)

(ex-2) def _get_data(**output_path="/tmp/wikipageviews.gz"**, **context):

- (ex-1) : 함수의 첫 인수인 output_path에 /tmp/wikipageviews.gz 설정됨
- (ex-1), (ex-2) : 결과 동일

(2) op_kwargs 인수

- "콜러블 함수"에 대한 "입력(인수값)"으로 "템플릿 문자열" 제공

(ex) def _get_data(**year, month, day, hour, output_path**, **_):

url=(

"https://dumps.wikimedia.org/other/pageviews/"

f"{year}/{year}-{month:0>2}/"

f"pageviews-{year}-{month:0>2}{day:0>2}-{hour:0>2}0000.gz"

)

request.urlretrieve(url, output_path)

get_data=**PythonOperator**(

task_id="get_data",

python_callable=_get_data,

op_kwargs={ // 사용자 정의 키워드 인수는

"year": "{{ execution_date.year }}", 콜러블 함수에 전달되기 전에

"month": "{{ execution_date.month }}", 템플릿 화 됨

```
            "day": "{{ execution_date.day }}",
            "hour": "{{ execution_date.hour }}",
            "output_path": " /tmp/wikipageviews.gz",
        },
        dag=dag,
)
```

#팀플릿의 인수 검사

- Airflow UI 이용해 "템플릿 인수 오류" 디버깅
- (UI기준) "**Rendered Template**" 버튼 클릭 -> 인수 검사
  - ➔ 렌더링 된 모든 연산자의 속성/값 표시
  - ➔ 검사하려면, Airflow에서 작업 스케줄해야 됨
  - ➔ CLI 사용해도 결과 동일 [CLI 사용시 메타 스토어에 아무것도 등록(X) -> 간편]

#다른 시스템과 연결하기

: Airflow는 태스크 간 데이터 전달하는 방법 2가지

(1) 메타스토어 사용해 태스크 간 결과 쓰고/읽음 (이를 **XCom** 이라 함)

- XCom이라는 기본 메커니즘 제공
- 메타스토어에서 선택 가능한 picklable 개체를 저장하고 나중에 읽을 수 있음
- **피클**(Pickle) : 파이썬의 직렬화 프로토콜

(참고: 직렬화 의미 - 메모리 개체 나중에 읽을 수 있도록 디스크에 저장할 수 있는 형식으로 변환하는 것)

- 크기 작은 데이터(객체) 전송 시, XCom 적합
- <피클링 불가 개체 경우> DB연결 or 파일 핸들러 방안 있음.

(2) 영구적 위치(ex: 디스크/DB)에 태스크 결과 기록

- 크기 큰 데이터(객체) 전송 시, 외부에 Data 유지하는 것이 적합
- **DB관련 Operator 클래스** 가져오기 위해 추가 패키지 설치 필요

(ex) pip install apache-airflow-providers-postgres

(ex) 쿼리생성(1) : PostgresOperator에 공급할 INSERT 구문 작성

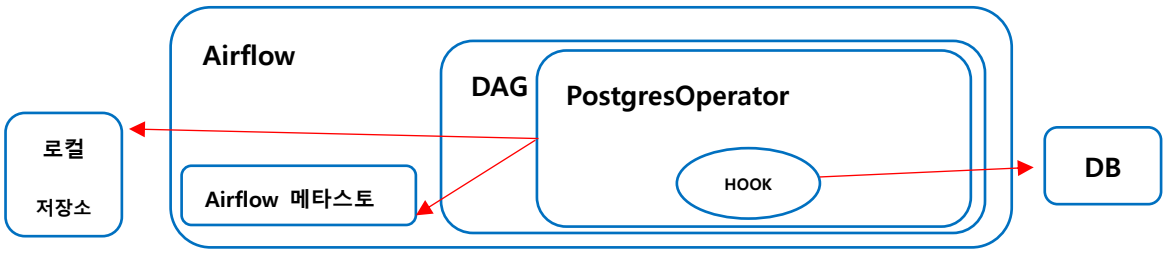
```
def _getch_pageviews(pagenames, execution_date, **_):
        result=dict.fromkeys(pagenames, 0)                    // 0으로 모든 페이지 뷰 결과 초기화
        with open("/temp/wikipageviews", "r") as f:
        for line in f:
                domain_code, page_title, view_counts, _=line.split("")
                if domain_code == "en" and page_title in pagenames:
                        result[page_title]=view_counts                  // 페이지 뷰 저장
        with open("/tmp/postgres_query.sql", "w") as f:
                for pagename, pageviewcount in result.items():        //각 결과에 대해 SQL 작성
                        f.write(
                                "INSERT INTO pageview_counts VALUES("
                                f" '{pagename}', {pageviewcount}', '{execution_date}' "
                                ");₩n)"
                                )
fetch_pageviews=PythonOperator(
        task_id="fetch_pageviews",
        python_callable=fetch_pageviews,
        op_kwargs={"pagenames": {"Google", "Amazon", "Apple", "Microsoft", "Facebook"}},
        dag=dag,
)
```

- 위 작업 실행시, PythonOperator에서 실행할 SQL 쿼리파일이 지정된 스케줄 간격으로 생성됨.

(ex) 연결(2) : PostgresOperator 호출

```
from airflow.providers.postgres.operators.postgres import PostgresOperator

dag = DAG(~~~, template_searchpath="/tmp")      // SQL 파일 경로탐색

write_to_postgres=PostgresOperator(

        task_id="write_to_postgres",

        postgres_conn_id="my_postgres",          // 연결에 사용할 식별ID

        sql="postgres_query.sql",                // SQL쿼리 or SQL쿼리 포함한 파일경로

        dag=dag,

)
```

- **template_searchpath** 인수 : INSERT ~ 문자열 외 "파일내용"도 템플릿 화 가능
  - ➔ 특정 확장자(ex: .sql) 파일 읽고 템플릿 화
  - ➔ 파일의 템플릿을 통해 표현된 후, PostsgresOperator에 의해 파일의 쿼리 실행
- (참고사항) PostgresOperator는 Postgre와 통신하기 위해 훅(hook) 이라는 것을 인스턴스화 함
  - ➔ 인스턴스화된 훅은 연결 생성, Postgre에 쿼리전송, 연결 종료 작업을 처리함
  - ➔ 여기서, Operator는 사용자 요청을 훅(hook)으로 전달하는 작업만 담당
  - ➔ 즉, 훅(hook)은 Operator 내부에서 동작함 -> 신경 쓸 필요 (x)



- PostgresOperator : **(1) postgres_conn_id="my_postgres", (2) sql="postgres_query.sql",** 의해 Postgre 연결 및 쿼리 실행 됨.

※ **지금까지 로직의 기반한 "그래프 뷰"**

- DAG가 매 시간 위키피디아 페이지 뷰 가져오고, 결과 Postgres에 기록함
- PostgresOperator는 Postgres DB에서 쿼리 실행 위해, 두개 인수 입력 필요
- DB 연결 설정 및 완료 후 연결 끊기 작업은 내부에서 처리됨
- **postgres_conn_id** 인수는 Postgres 자격증명 식별자

(ex) CLI 이용해 Airflow 자격증명 저장

```
airflow connections add ₩
    --conn-type postgres ₩
    --conn-host localhost ₩
    --conn-login postgres ₩
    --conn-password mysecretpassword ₩
    my_postgres                                    // 연결 식별자
```

- Airflow UI : **Admin > Connections** 클릭 -> 저장된 모든 연결 확인가능