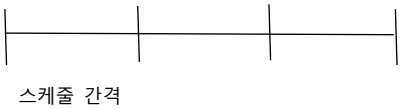
Chapter 3 : Airflow의 스케줄링

#스케줄 개념

- (스케줄 간격 단위) DAG실행
- (스케줄 간격 단위) 증분 데이터 처리
- 백필 : 과거 특정시점 작업 가능 (과거 데이터 공백 채움)



[ex] 사용자이벤트 처리하기

- 로직구성 : (1)데이터 가져오기 + (2)통계 및 계산
  - (1)경우, BashOperator 이용
  - (2)경우, PythonOperator 사용해 Pandas DataFrame에 데이터 로드 후, groupby / aggregation 사용해 계산(개수 확인)

[ex] DAG 객체 인스턴스 생성

```
dag = DAG(
        dag_id = "01_unscheduled",
        schedule_interval = None,                    //스케줄 간격설정
        start_date = dt.datetime(2019, 1, 1),        //DAG 시작날짜
        end_date = dt.datetime(year=2019, month=1, day=5),    //DAG 종료날짜
)
```

- **schedule_interval = None**

  : DAG 예약실행(X), UI or API로 수동으로 트리거 발생시켜 실행

  - 1회만 : @once

- DAG는 2019-01-02 최초 실행 됨.
- 시작날짜

  - start_date = dt.datetime(2019, 1, 1)

  - start_date = dt.datetime(year=2019, month=1, day=1)

  (참고 : 종료날짜 미 설정 시, DAG 지속적으로 실행됨)

#스케줄 간격 설정 방법

(1) 프리셋

(사용 예시) schedule_interval = "@프로셋_명"

- 매일 자정(1회) : **@daily**
- 매시간 변경시 (1회) : **@hourly**
- 매주 일요일 자정(1회) : **@weekly**
- 매월 1일 자정 (1회) : **@monthly**
- 매년 1월 1일 자정 (1회) : **@yearly**

(2) Cron기반

: 유닉스(UNIX) 기반 운영체제에서 주기적으로 반복되는 작업(스케줄링된 작업)을 자동으로 실행할 수 있도록 해주는 작업 스케줄링 도구

- 주로 서버 환경에서 사용되며, 정해진 시간과 날짜에 프로그램을 실행하거나 스크립트를 수행할 때 사용됩니다. Cron은 "**cron job**"이라고 불리는 개별 작업을 정의하고 관리
- (특징_1)자동화된 작업 스케줄링 : Cron을 사용하면 작업을 특정 시간, 요일, 월, 분 단위로 정기적으로 실행 가능
- (특징_2)유연한 스케줄링 : 다양한 스케줄링 옵션을 통해 일별, 주별, 월별, 또는 특정 요일/시간에 작업을 수행 가능
- (특징_3)시스템 레벨 작업 관리 : 시스템 레벨에서 설정된 cron job은 시스템 재부팅 후에도 자동으로 다시 설정되어 작업을 이어나갈 수 있음
- 유닉스 및 리눅스 환경에서 필수적인 도구
- (실행방법) cron job : "시간/날짜가 필드 값 = 시스템 시간 경우 실행"
- (단점) '특정 빈도' 스케줄링 정의 안됨 (ex: 3일에 한번 실행)
- (Cron 설정 파일) crontab
- Cron 형식

```
*     *     *     *     *  command to be executed
|     |     |     |     |
|     |     |     |     +----- 요일 (0 - 7) (일요일은 0과 7)
|     |     |     +------- 월 (1 - 12)
|     |     +---------- 일 (1 - 31)
|     +------------ 시간 (0 - 23)
+-------------- 분 (0 - 59)
```

- (예시) 매일 오후 2시 30분에 backup.sh 스크립트를 실행
  - 30 14 * * * /home/user/backup.sh

- (예시) 매시간(정시)
  - 0 * * * * /home/user/backup.sh
- (예시) 매일(자정)
  - 0 0 * * * /home/user/backup.sh
- (예시) 매주(일요일 자정)
  - 0 0 * * 0 /home/user/backup.sh
- (예시) 매월 1일 자정
  - 0 0 1 * * /home/user/backup.sh
- (예시) 매주 토요일 23시 45분
  - 45 23 * * SAT /home/user/backup.sh
- (예시) 매주 월, 화, 금 자정
  - 0 0 * * MON, WED, FRI /home/user/backup.sh
- (예시) 매주 월~금 자정
  - 0 0 * * MON-FRI /home/user/backup.sh
- (예시) 매일 자정 및 오후 12시
  - 0 0,12 * * * /home/user/backup.sh
- crontab -e 명령어 : crontab 파일 편집
- crontab -l 명령어 : cron 작업 목록 보기
- crontab -r 명령어 : cron 작업을 제거

(3) 빈도기반

: timedelta(표준 라이브러리인 datetime 모듈에 포함된) 인스턴스 사용

```
dag = DAG(
        dag_id = "04_time_delta",
        schedule_interval = dt.timedelta(days=3),                    //스케줄 간격설정
        start_date = dt.datetime(year=2019, month=1, day=1),         //DAG 시작날짜
        end_date = dt.datetime(year=2019, month=1, day=5),           //DAG 종료날짜
)
```

- 3일마다 실행 : **timedelta(days=3)** (2019-1-4 / 2019-1-7 ……)
- 10분마다 실행 : **timedelta(minutes=10)**
- 2시간마다 실행 : **timedelta(hours=2)**

#데이터 증분 처리 (배시 명령어 수정)

- 매일 전체 데이터 다운로드 + 계산 -> 비효율적
- DAG 수정 (데이터 순차적으로 가져오도록) -> 스케줄 간격에 해당하는 일자 이벤트만 다운로드 + 계산(통계)
  - ➢ DAG수정 : 특정 날짜 데이터 다운로드
    - ⇨ 시작 및 종료날짜 '매개변수' 함께 정의 -> 해당 날짜 데이터 가져오도록 API 호출 조정
    - ⇨ (ex) 배시 명령어
      curl -0 **http://localhost:5000/events?start_date=2019-01-01&end_date=2019-01-01**
    - ⇨ start_date : 포함날짜 / end_date : 포함하지(x) 날짜
    - ⇨ 2019-01-01 00:00:00 ~ 2019-01-01 23:59:59 사이 발생한 데이터 가져옴
- 날짜별 분리된 단일파일로 저장 [API가 제한하고 있는 파일로 저장(X) : (ex) 30일 저장]
  - ➢ 시간 지남에 따라 매일 순차적으로 파일 저장

[ex] 특정 시간 간격에 대한 이벤트 가져오기

```
fetch_events=BashOperator(
        task_id="fetch_events",
        bash_command=(
            "mkdir -p /data && "
            "curl -o /data/events.json "
            "http://localhost:5000/events?"
            "start_date=2019-01-01&"
            "end_date=2019-01-02"
        ),
        dag=dag,
)
```

#실행날짜 사용해 동적 시간 참조

[1] 동적 매개변수

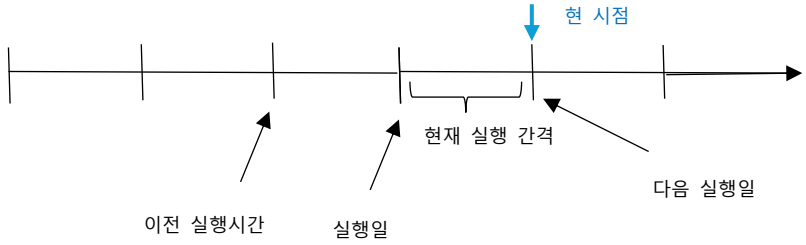
목적 : 스케줄 간격 정의

사용 : Operator에 참조해 사용 (배시 명령이 실행될 날짜를 동적으로 포함시켜 사용)

- **execution_date**
  - ➢ 스케줄 간격의 시작시간 (DAG 실행되는 날짜/시간 나타냄)

  (참고: 특정 날짜가 아닌, 스케줄 간격으로 실행되는 시작 시간 나타내는 '타임스탬프')
- **next_execution_date**
  - ➢ 스케줄 간격의 종료시간
- **previous_execution_date**
  - ➢ 과거의 스케줄 간격의 시작시간
  - ➢ 현재 시간간격의 데이터 vs 이전 간격의 데이터 비교분석에 유용

[참고] Airflow의 실행 날짜 개념



[ex] 특정 날짜 지정 위해 템플릿

```
fetch_events=BashOperator(
        task_id="fetch_events",
        bash_command=(
            "mkdir -p /data && "
            "curl -o /data/events.json "
            "http://localhost:5000/events?"
            "start_date={{execution_date.strftime('%Y-%m-%d')}}"
            "&end_date={{next_execution_date.strftime('%Y-%m-%d')}}"
        ),
```

```
        dag=dag,
    )
```

- Jinja 템플릿으로 형식화된 execution_date 삽입
  {{ }} 구문
- datetime의 strftime 메서드 사용 : 문자열 형식으로 반환
  (두 실행날짜 모두 datetime 개체이므로)

[2] [1]에 대한 축약 매개변수

- ds 및 ds_nodash : YYYY-MM-DD 및 YYYYMMDD 형식으로 된 execution_date의 다른표현

- **ds** or **ds_nodash** = execution_date
- **next_ds** or **next_ds_nodash** = next_execution_date
- **pre_ds** or **pre_ds_nodash** = previous_execution_date

[ex] 템플릿에서 축약어 사용

```
fetch_events=BashOperator(
        task_id="fetch_events",
        bash_command=(
            "mkdir -p /data && "
            "curl -o /data/events.json "
            "http://localhost:5000/events?"
            "start_date={{ds)}}"
            "end_date={{next_ds)}}"
        ),
        dag=dag,
)
```

#데이터 파티셔닝

: 데이터 세트를 관리하기 쉽도록 작은 조각으로 나누는 작업

(일반적으로) 스케줄 간격으로 데이터 다운시, 각각 새로운 태스크가 전일 데이터 덮어씌움.

(해결방법) events.json 파일에 새 이벤트 추가 -> 하나의 JSON파일에 모든 데이터 작성 가능

(단점) 특정 날짜 계산(통계) 하려면, 모든 데이터 다운. + 장애시, 모든 데이터 영향 위험.

⇩

(해결방법) 데이터 출력을 실행 날짜 이름의 파일에 기록 -> 즉, 데이터 세트를 일일배치로 나눔.

- "curl -o **/data/events/{{ds}}.json** " // 반환 값이 템플릿 파일 이름에 기록됨.
- 2019-01-01 실행 날짜에 다운로드되는 모든 데이터, **/data/events/2019-01-01.json** 파일에 기록됨.

[기존로직] 이벤트 통계작업

```
def _calculate_stats(input_path, output_path):
    """데이터 통계 계산"""
    Path(output_path).parent.mkdir(exist_ok=True)
    events=pd.read_json(input_path)
    stats=events.groupby(["date", "user"]).size().reset_index()
    stats.to_csv(output_path, index=False)


calculate_stats=PythonOperator(
    task_id="calculate_stats",
    python_callable=_calculate_stats,
    op_kwargs={
        "input_path": "/data/events.json",
        "output_path": "/data/events.csv",
    },
    dag=dag,
```

)

- 위 로직은, 매일 전체 데이터 세트 로드, 전체 데이터 계산(통계)

[변경로직] 실행 스케줄 간격마다 통계 계산

```
def _calculate_stats(**context):                                    //모든 context변수 수신
        input_path=context["templates_dict"]["input_path"]          // templates_dict 개체에서
        output_path=context["templates_dict"]["output_path"]          템플릿 값 검색
        """ 아래 로직은 기존 로직과 동일 """
        Path(output_path).parent.mkdir(exist_ok=True)
        events=pd.read_json(input_path)
        stats=events.groupby(["date", "user"]).size().reset_index()
        stats.to_csv(output_path, index=False)


calcutate_stats=PythonOperator(
        task_id="calculate_stats",
        python_callable=_calculate_stats,
        templates_dict={
        "input_path": "/data/events/{{ds}}.json",                   // 템플릿 되는 값 전달
        "output_path": "/data/events/{{ds}}.csv",
        },
        dag=dag,
)
```

- 입,출력 경로 변경해 파티션된 데이터 세트 사용 (**입,출력 경로 템플릿화**)
  (보일러플레이트 코드로 작성)
- templates_dict 매개변수 사용해 템플릿화.
- context개체에서 함수 내부 템플릿 값 확인 가능.

위 로직으로 "데이터의 작은 서브 세트만 처리" -> 점진적 통계 계산

#Airflow 실행날짜 이해

- 시작날짜 / 스케줄 간격 / 종료날짜 ---3가지 매개변수 이용해 ---> DAG 실행시점 제어
- Airflow 시간처리는 "스케줄 간격"에 따라 실행됨.
- Airflow의 실행날짜 = DAG의 시작날짜로 정의
  (참고) Airflow : 간격기반 스케줄링 윈도우 / cron : 시점 기반 시스템에서 파생된 윈도우

#백필(Backfilling)

: 과거 시점 지정해 DAG실행 (과거 데이터 세트 로드/분석 목적)

- 아직 실행되지 않은 과거 스케줄 간격 예약/실행
- 과거 시작 날짜 지정 후 해당 DAG 활성화하면, 현재 시간 이전 ~ 과거 시작 이후의 모든 스케줄 간격 생성됨.
  - 위 동작은 DAG의 catchup 매개변수 의해 제어
  - false 설정 = 비활성화 의미.

[ex] 과거 시점 태스크 실행을 피하기위해 catchup 비활성화

```
dag=DAG(

        dag_id="09_no_catchup",

        schedule_interval="@daily",

        start_date=dt.datetime(year=2019, month=1, day=1),

        end_date=dt.datetime(year=2019, month=1, day=5),

        catchup=False,

)
```

  - **catchup=False** 설정으로 과거 모든 스케줄 간격으로 태스크 실행 (X)
    최근 스케줄 간격에 대해서만 실행됨.
  - Catchup 기본값은 Airflow 구성파일(configuration file)에서 catchup_by_default 값 설정으로 제어
  - API 상 과거 데이터를 30일 범위만 기록/제공하는 경우, 그 이전 데이터 백필 수행하더라도 데이터 로드 (X)
  - 함수(계산로직) 수정 후, 과거 데이터 새로운 산식으로 분석 원하는 경우,
    이전 실행 태스크의 일부로 이전 데이터 파티션을 이미 저장했기 때문에 30일 제한 관계없이 정상적으로 백필 실행 가능.

#Airflow 태스크 핵심속성

- **원자성**(atomicity) : Airflow 태스크는 성공적으로 수행 후 결과 생성 or 시스템 영향 없도록 실패 정의해야 함 (All or Nothing / DB 트랜잭션 특성과 유사)
- **멱등성**(idempotency) : 동일한 입력(값)으로 동일한 태스크 여러 번 호출 시, 실행 횟수 상관없이 동일한 결과(값) 나와야 함.
  - 기존 결과 확인 or 이전 태스크 결과 덮어쓸지 여부 확인 -> 멱등성 유지