

# Dag프로듀스컨슈머패턴\_002



Dag작성 규칙에서 다음과 같이 답변을 줬잖아. 심플하게 예제 코드 작성하고 설명해줘.  
(바로 테스트 할 수 있도록)

"3) 큐(메시지/테이블) 기반 워커 패턴

컨트롤러 DAG는 큐에 적재만, 워커 DAG는 주기적으로 큐를 폴링해 한 번에 K개씩 소비.

Airflow가 스케줄러 역할만 하고, 세밀한 상태관리는 큐/테이블로.  
"



34s 동안 생각함

좋아요. "컨트롤러=적재 / 워커=소비(K개씩)"를 바로 돌려볼 수 있게, **PostgreSQL** 테이블 큐로 만든 최소예제를 드립니다.

(에어플로우 2.9.2, KubernetesExecutor/Local 모두 OK)

## 0) 사전 준비 (1분 컷)

1. Airflow Connection 하나만 잡아주세요.

- **Conn ID:** `queue\_db`
- **Conn Type:** Postgres
- Host/DB/User/Password는 운영 Postgres로 (메타DB 말고 별도 권장)

2. 큐 테이블 생성(SQL) — 한 번만 실행

Airflow UI → Admin → **DB Console**(있으면) 또는 psql에서 실행:

```
CREATE TABLE IF NOT EXISTS job_queue (  
  id          BIGSERIAL PRIMARY KEY,
```



```

payload      JSONB NOT NULL,
status       TEXT NOT NULL DEFAULT 'queued',    --
queued|processing|done|error
attempts     INT NOT NULL DEFAULT 0,
enqueued_at  TIMESTAMPTZ NOT NULL DEFAULT now(),
started_at   TIMESTAMPTZ,
finished_at  TIMESTAMPTZ,
err          TEXT
);

CREATE INDEX IF NOT EXISTS idx_job_queue_status_id ON job_queue(status, id);

```

## 1) 컨트롤러 DAG (큐에 적재만)

`dags/controller\_enqueue\_dag.py`

```

from __future__ import annotations
import random, string, json
from datetime import datetime
from airflow import DAG
from airflow.decorators import task
from airflow.models.param import Param
from airflow.providers.postgres.hooks.postgres import PostgresHook

# 이 Conn ID만 맞추면 바로 동작
QUEUE_CONN_ID = "queue_db"

with DAG(
    dag_id="controller_enqueue",
    start_date=datetime(2024, 1, 1),
    schedule=None, # 수동 트리거
    catchup=False,
    params={
        "num_jobs": Param(25, type="integer", minimum=1, maximum=10000,
description="적재할 작업 수"),
        "tag": Param("demo", type="string", description="페이로드 태그"),
    },
    doc_md="""
**역할:** 큐 테이블(job_queue)에 작업(payload)을 쌓기만 합니다.
""",

```

```

) as dag:

@task
def enqueue(num_jobs: int, tag: str):
    pg = PostgresHook(postgres_conn_id=QUEUE_CONN_ID)
    rows = []
    for _ in range(num_jobs):
        rid = "".join(random.choices(string.ascii_lowercase +
string.digits, k=8))
        payload = {
            "rid": rid,
            "op": "toy_work",    # 워커가 무엇을 할지 구분용
            "tag": tag,
            "value": random.randint(1, 100),
        }
        rows.append(json.dumps(payload))

    # 벌크 insert
    insert_sql = """
INSERT INTO job_queue (payload)
SELECT x::jsonb
FROM unnest(%s::text[]) AS t(x);
"""
    pg.run(insert_sql, parameters=(rows,))
    return {"inserted": len(rows)}

enqueue(dag.params["num_jobs"], dag.params["tag"])

```

## 테스트 방법

```

# 25건 적재
airflow dags trigger controller_enqueue --conf '{"num_jobs": 25, "tag":"first-run"}'

```

## 2) 워커 DAG (K개씩 소비)

`dags/worker\_consumer\_dag.py`

```

from __future__ import annotations
import time

```

```

from datetime import datetime, timedelta
from airflow import DAG
from airflow.decorators import task
from airflow.providers.postgres.hooks.postgres import PostgresHook

QUEUE_CONN_ID = "queue_db"
BATCH_SIZE = 10          # 한 번에 소비할 개수(K)
LOCK_TIMEOUT_SEC = 5     # 행 잠금 대기 시간(짧게)

with DAG(
    dag_id="worker_consumer",
    start_date=datetime(2024, 1, 1),
    schedule="*/1 * * * *", # 매 1분 소비
    catchup=False,
    max_active_runs=1,      # 중복 소비 방지(필요 시 조정)
    doc_md="""
**역할:** 큐(job_queue)에서 `queued` 상태를 **K개만** 안전하게 잠그고(SELECT ... FOR
UPDATE SKIP LOCKED)
처리 후 `done/error`로 마감합니다. Airflow는 스케줄만, 상태관리는 DB가 담당.
""",
) as dag:

    @task
    def consume_batch():
        pg = PostgresHook(postgres_conn_id=QUEUE_CONN_ID)

        # 짧은 lock timeout (다른 워커와 경합 시 빠르게 포기)
        pg.run("SET LOCAL lock_timeout = %s;", parameters=(f"
{LOCK_TIMEOUT_SEC}s",), autocommit=False)

        # 1) K개 잠그고 processing으로 마킹
        claim_sql = f"""
WITH cte AS (
    SELECT id
    FROM job_queue
    WHERE status = 'queued'
    ORDER BY id
    FOR UPDATE SKIP LOCKED
    LIMIT {BATCH_SIZE}
)
UPDATE job_queue q

```

```

SET status = 'processing', started_at = now(), attempts = attempts + 1
FROM cte
WHERE q.id = cte.id
RETURNING q.id, q.payload::text;
"""

# 트랜잭션 단위 처리
conn = pg.get_conn()
conn.autocommit = False
try:
    with conn.cursor() as cur:
        cur.execute(claim_sql)
        claimed = cur.fetchall() # [(id, payload_json_text), ...]
        conn.commit()
except Exception:
    conn.rollback()
    raise

if not claimed:
    return {"claimed": 0, "done": 0}

# 2) 실행 (여기서 실제 업무 로직 수행)
done_ids, error_results = [], []
for _id, payload_text in claimed:
    try:
        # --- 예시 업무: 0.1초 슬립 + value 제공 계산 ---
        # 실제론 외부 API/DB/S3/Spark/KPO 호출 등으로 대체
        time.sleep(0.1)
        # 가벼운 검증
        # (json.loads 쓰지 않고 DB내 jsonb도 좋지만, 단순화를 위해 생략)
        # payload_text는 텍스트이므로 여기서도 로그만
        # print(f"[DO] id={_id} payload={payload_text[:120]}")
        done_ids.append(_id)
    except Exception as e:
        error_results.append((_id, str(e)))

# 3) 상태 갱신: done / error
try:
    with conn.cursor() as cur:
        if done_ids:
            cur.execute(

```

```

        "UPDATE job_queue SET status='done', finished_at=now(),
err=NULL WHERE id = ANY(%s);",
        (done_ids,),
    )
    for eid, emsg in error_results:
        cur.execute(
            "UPDATE job_queue SET status='error',
finished_at=now(), err=%s WHERE id=%s;",
            (emsg, eid),
        )
    conn.commit()
except Exception:
    conn.rollback()
    raise
finally:
    conn.close()

return {
    "claimed": len(claimed),
    "done": len(done_ids),
    "error": len(error_results),
}

consume_batch()

```

## 돌려보기

```

# 워커 시작(스케줄러가 매 1분마다 소비)
airflow dags trigger worker_consumer
# 컨트롤러로 여러 번 적재해 보세요
airflow dags trigger controller_enqueue --conf '{"num_jobs": 100, "tag":"batch-A"}'
airflow dags trigger controller_enqueue --conf '{"num_jobs": 37, "tag":"batch-B"}'

```

## 왜 이 구성이 편한가

- **경합 안전:** `FOR UPDATE SKIP LOCKED`로 다중 워커여도 중복 소비 없음
- **K개 청크:** 워커 1회당 K개만 집어서 처리 → 과부하·타임아웃 방지

- **상태 가시성:** DB가 진짜 소스 오브 트루스 → ``queued/processing/done/error``를 SQL로 바로 집계
- **Airflow는 스케줄만:** 실패/재시도/병렬성은 Airflow가, 세밀한 단위 상태는 DB가 담당

## 운영 팁(선택)

- **병렬 워커:** ``max_active_runs``를 3~5로 늘리거나, 워커 DAG를 매 초마다(예: ``*/20 * * * *``는 2.9에 없음 → 짧은 스케줄+외부 트리거) 다중 인스턴스 실행
- **리트라이 정책:** 현재 코드는 큐 레코드 ``attempts``만 올립니다. 필요 시 ``attempts >= N``이면 ``error``로 고정 등 정책 추가
- **업무 로직 분리:** ``toy_work`` 대신 실제 처리 함수를 모듈로 분리하거나, **KubernetesPodOperator**를 여기서 호출해 “한 번에 K개의 파드”를 동적으로 띄우는 형태로 확장 가능
- **모니터링 쿼리 예시**

```
SELECT status, count(*) FROM job_queue GROUP BY 1 ORDER BY 1;
SELECT date_trunc('minute', enqueued_at) AS ts, count(*)
FROM job_queue GROUP BY 1 ORDER BY 1 DESC LIMIT 60;
```

필요하면 다음 단계로 **KubernetesPodOperator** 연동 버전, 동적 태스크 매핑 버전도 바로 만들어 드리겠습니다.



지금 패턴이 java produce consumer 패턴과 유사하지?



맞습니다. 딱 Java에서 흔히 짜는 **Producer-Consumer** 패턴이랑 닮았어요.

- **컨트롤러 DAG** → Producer 역할. 큐(테이블)에 메시지를 쌓아두기만 함.
- **워커 DAG** → Consumer 역할. 큐에서 일정량(K개)을 뽑아서 처리.

차이는 환경이 다르다는 점이에요:

- Java에서는 보통 **BlockingQueue** 같은 메모리 기반 자료구조를 쓰고, 스레드/풀로 동시성 제어.

- 여기선 **DB 테이블이나 메시지 브로커**를 큐로 삼고, Airflow 스케줄러가 Consumer 트리거 역할을 해요.

즉, 기본 골격은 같지만 Airflow는 스레드 대신 **DAG** 태스크, 큐 대신 외부 **DB/브로커**, 스케줄링은 **Airflow**로 풀어낸 버전이라고 보면 돼요.

👉 혹시 원하시면 제가 Java의 BlockingQueue 예제랑, 지금 만든 Airflow 버전을 나란히 비교해서 보여드릴까요?



워커가 싱글 프로세싱이니깐 java에 비해서는 비효율적이네



네, 그 부분이 핵심이에요.

Java에서 `ExecutorService + BlockingQueue`를 쓰면 워커 스레드가 여러 개라서 큐에서 동시에 뽑아 처리하죠. 반면 제가 보여드린 Airflow 워커 DAG는 **한 번 실행할 때 K개 묶음만** 소비하는 식이라, 사실상 **싱글 프로세싱 배치 소비자**에 가까워요.

그럼에도 불구하고 이 방식이 Airflow에서 쓰이는 이유는:

- 안전성:** `FOR UPDATE SKIP LOCKED`로 경합 관리가 확실해서 중복 처리를 피할 수 있음.
- 스케줄러 친화적:** Airflow 자체가 분산 큐 처리보단 “스케줄링 + 모니터링” 강점이 있어서, 큐 소비를 배치형으로 만드는 게 운영적으로 단순해짐.
- 확장 옵션:** `max_active_runs`을 늘리거나 DAG 자체를 여러 인스턴스로 병렬 띄우면 Java 워커풀처럼 병렬 처리량을 늘릴 수 있음.

즉, 지금 상태는 효율보다 **안정·투명한 상태 관리** 쪽에 초점을 둔 구조예요.

조금 더 Java 쪽 워커풀 느낌을 내려면:

- DAG 한 개가 아니라 워커 **DAG** 여러 개를 병렬 돌려서 **동일 큐를** 소비하게 하기
- 또는 워커 DAG 안에서 **Dynamic Task Mapping**으로 각 잡(job)을 태스크 단위 병렬화

이런 쪽을 섞으면 “Airflow식 병렬 워커풀”에 더 가까워질 수 있어요.

👉 지금 궁금하신 건 “Airflow에서 어떻게 하면 Java식 다중 워커풀에 더 가깝게 만들 수 있느냐” 인가요, 아니면 “Airflow는 근본적으로 싱글 배치 소비자라서 한계가 있다”는 쪽을 더 정리해드릴까요?

