**클라우드 기반 빅데이터 솔루션**

**AWS EMR(Elastic MapReduce) Serverless Workshop 실습**

**AWS EMR(Amazon Elastic MapReduce)**

**1. 개요**

Amazon EMR은 대규모 데이터 처리 및 분석을 위한 클러스터 기반의 분산 컴퓨팅 플랫폼이다. Apache Hadoop, Spark, Hive, HBase, Presto 등 오픈소스 빅데이터 프레임워크를 지원하며, 대량의 데이터를 쉽고 빠르게 처리할 수 있도록 AWS에서 완전관리형으로 제공한다.

**2. 주요 특징**

| **특징** | **설명** |
| --- | --- |
| **완전관리형** | 클러스터 생성, 구성, 확장, 종료를 자동화 |
| **유연한 프레임워크 지원** | Hadoop, Spark, Hive, HBase, Presto 등 사용 가능 |
| **자동 확장** | 작업 부하에 따라 노드 수를 자동 조정 가능 |
| **저렴한 비용** | EC2 스팟 인스턴스를 사용해 비용 절감 가능 |
| **S3 연동** | 입력/출력 데이터를 Amazon S3에서 직접 처리 가능 |

**3. 클러스터 구성 요소**

| **구성 요소** | **설명** |
| --- | --- |
| **마스터 노드 (Master Node)** | 클러스터의 상태를 관리하고 작업을 분산함 |
| **코어 노드 (Core Node)** | HDFS에 데이터를 저장하고, 작업을 수행함 |
| **태스크 노드 (Task Node)** | 계산 작업만 수행, HDFS는 사용하지 않음 |

**4. EMR 클러스터 생성 예시 (AWS CLI)**

aws emr create-cluster \

--name "MyEMRCluster" \

--release-label emr-6.14.0 \

--applications Name=Spark Name=Hive \

--ec2-attributes KeyName=myKey \

--instance-type m5.xlarge \

--instance-count 3 \

--use-default-roles

**5. 클러스터 관리 명령어**

# 클러스터 목록 조회

aws emr list-clusters

# 클러스터 상세 조회

aws emr describe-cluster --cluster-id j-XXXXXXXXXXXXX

# 클러스터 종료

aws emr terminate-clusters --cluster-ids j-XXXXXXXXXXXXX

**6. 작업 제출 방식**

**6.1 스텝(Step) 방식**

EMR에 작업 단위(Step)를 제출하면 순차적으로 실행된다.

aws emr add-steps \

--cluster-id j-XXXXXXXXXXXXX \

--steps Type=Spark,Name="MySparkJob",ActionOnFailure=CONTINUE,Args=[--class,org.apache.spark.examples.SparkPi,s3://my-bucket/spark-examples.jar,10]

**6.2 EMR Studio 또는 Notebook 사용**

EMR Studio를 통해 웹 기반 인터페이스로 Spark/Hive 작업을 실행할 수 있다.

**7. S3와의 연동**

* 데이터 소스와 결과물은 HDFS 대신 S3에 저장 가능
* Spark SQL 예시:

CREATE EXTERNAL TABLE logs (

id STRING,

message STRING

)

STORED AS PARQUET

LOCATION 's3://my-bucket/log-data/';

**8. 자동 확장**

EMR 클러스터는 Amazon CloudWatch와 연동하여 오토스케일링 정책 설정 가능

aws emr put-auto-scaling-policy \

--cluster-id j-XXXXXXXXXXXXX \

--instance-group-id ig-XXXXXXXXXXXXX \

--auto-scaling-policy file://policy.json

policy.json 예시는 다음과 같다:

{

"Constraints": {

"MinCapacity": 2,

"MaxCapacity": 10

},

"Rules": [

{

"Name": "ScaleOutMemory",

"Action": {

"SimpleScalingPolicyConfiguration": {

"AdjustmentType": "CHANGE\_IN\_CAPACITY",

"ScalingAdjustment": 1,

"CoolDown": 300

}

},

"Trigger": {

"CloudWatchAlarmDefinition": {

"ComparisonOperator": "GREATER\_THAN",

"EvaluationPeriods": 1,

"MetricName": "MemoryAvailableMB",

"Namespace": "AWS/ElasticMapReduce",

"Period": 300,

"Statistic": "AVERAGE",

"Threshold": 5000,

"Unit": "Megabytes"

}

}

}

]

}

**9. 보안**

* IAM 역할 기반 권한 제어
* EC2 키페어를 통한 SSH 접속
* 데이터 암호화: S3-SSE, EMRFS 암호화, TLS 등
* Kerberos를 통한 인증 설정 지원

**10. CloudFormation 예시**

Resources:

EMRCluster:

Type: AWS::EMR::Cluster

Properties:

Name: MyEMRCluster

ReleaseLabel: emr-6.14.0

Applications:

- Name: Spark

Instances:

Ec2KeyName: myKey

InstanceCount: 3

KeepJobFlowAliveWhenNoSteps: true

MasterInstanceType: m5.xlarge

CoreInstanceType: m5.xlarge

CoreInstanceCount: 2

JobFlowRole: EMR\_EC2\_DefaultRole

ServiceRole: EMR\_DefaultRole

**11. 요약**

* Amazon EMR은 분산처리 기반의 빅데이터 분석 플랫폼이다.
* Hadoop, Spark, Hive 등의 오픈소스를 간편히 실행할 수 있다.
* S3, IAM, CloudWatch, CloudFormation 등 AWS 서비스와 밀접하게 통합된다.
* 자동 확장, 저비용 운영, 유연한 작업 제출 방식이 주요 장점이다.

**12. 참고 명령어 정리**

# 클러스터 생성

aws emr create-cluster --name "MyCluster" --release-label emr-6.14.0 --applications Name=Spark ...

# 클러스터 종료

aws emr terminate-clusters --cluster-ids j-XXXXXXXXXXXXX

# Spark 작업 추가

aws emr add-steps --cluster-id j-XXXX --steps Type=Spark,Name=Job,...

# 오토스케일 정책 추가

aws emr put-auto-scaling-policy --cluster-id ... --instance-group-id ... --auto-scaling-policy file://policy.json

**[실습 시작]**

아래 Workshop 실습을 수행한다

https://catalog.us-east-1.prod.workshops.aws/workshops/e8e8fbb5-c3fb-4f86-bf77-0ba1fe402c55/en-US

**EMR 서버리스 워크숍**

텍스트, 스크린샷, 번호, 폰트이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

지시대로 Workshop 실습을 수행한다

**옵션 1 - 콘솔에서 Spark 작업 제출**

Job 실행 후 S3 출력 결과 확인 방법

Amazon S3->버킷->emrserverless-workshop-891377038690->wordcount\_output/

으로 들어가서 xxxxxxxxx.csv 파일을 하나 선택(첫번째)하여 클릭한다

우측 상단의 [객체 작업]을 클릭해서 **S3 Select를 사용한 쿼리**를 클릭한다

텍스트, 스크린샷, 폰트, 번호이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

CSV 구분 기호를 사용자 지정으로 선택하고 사용자 지정 CSV 구분 기호를

큰 따옴표(”)로 입력한다음 그대로 SQL 쿼리 실행을 누른다

텍스트, 스크린샷, 폰트이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다. 텍스트, 폰트, 로고, 상징이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

텍스트, 스크린샷, 소프트웨어, 폰트이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

**옵션 2 - CLI에서 Spark 작업 제출** (약 4분 36초 소요)

<스크립트 예시>

|  |
| --- |
| export JOB\_ROLE\_ARN=arn:aws:iam::891377038690:role/EMRServerlessS3RuntimeRole  export S3\_BUCKET=s3://emrserverless-workshop-891377038690  export APPLICATION\_ID=00frebt1ct62mm09  aws emr-serverless start-job-run --application-id ${APPLICATION\_ID} --execution-role-arn ${JOB\_ROLE\_ARN} --name "Spark-WordCount-CLI" --job-driver '{  "sparkSubmit": {  "entryPoint": "s3://us-east-1.elasticmapreduce/emr-containers/samples/wordcount/scripts/wordcount.py",  "entryPointArguments": [  "'"$S3\_BUCKET"'/wordcount\_output/"  ],  "sparkSubmitParameters": "--conf spark.executor.cores=1 --conf spark.executor.memory=4g --conf spark.driver.cores=1 --conf spark.driver.memory=4g --conf spark.executor.instances=1 --conf spark.hadoop.hive.metastore.client.factory.class=com.amazonaws.glue.catalog.metastore.AWSGlueDataCatalogHiveClientFactory"  }  }' |

실행 상태 확인 : 새로운 Job으로 실행된다

텍스트, 스크린샷, 폰트, 번호이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

앞과 동일하게 S3 Select를 사용한 쿼리를 확인해본다

**EMR Serverless에서 사용자 정의 이미지 사용**

다운 받은 custom-image-resources.zip 파일 업로드 경로 :

/home/ec2-user/environment/ 아래에 업로드 (최 상위 폴더임)

**Submit Hive jobs from EMR Studio**

텍스트, 폰트, 노랑, 로고이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다. 을 클릭해서 job 생성

Job configuration

|  |
| --- |
| {  "applicationConfiguration": [  {  "classification": "hive-site",  "configurations": [],  "properties": {  "hive.exec.scratchdir": "s3://**emrserverless-workshop-891377038690**/hive/scratch",  "hive.metastore.warehouse.dir": "s3://**emrserverless-workshop-891377038690**/hive/warehouse"  }  }  ]  } |

Application logs and metrics -> Upload logs to your Amazon S3 bucket->

Amazon S3 storage location URI :

**s3://aws-data-analytics-workshops/emr-serverless-workshop/log**

으로 설정하면 다음과 같은 오류 발생함

텍스트, 스크린샷, 폰트, 번호이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

emrserverless-workshop-891377038690 버킷 안에 미리 log 폴더를 미리 만들어 놓고

Amazon S3 storage location URI에서

**s3://emrserverless-workshop-891377038690/log**

으로 선택해주고 **[Submit job run]**을 클릭한다

텍스트, 폰트, 라인, 스크린샷이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

실행 후 결과

텍스트, 폰트, 번호, 라인이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

출력 데이터 설명

텍스트, 스크린샷, 번호, 폰트이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

뉴욕시 택시 운행 기록(NYC Taxi Trips)의 일부로 , 각 열은 택시 운행의 세부 정보를 담고 있다

**🟨 주요 컬럼 설명**

| **컬럼명** | **설명** |
| --- | --- |
| **vendorid** | 택시 회사 ID.  1: Creative Mobile Technologies (CMT) 2: VeriFone Inc. |
| **tpep\_pickup\_datetime** | 승차 시각 (pickup 시점) |
| **tpep\_dropoff\_datetime** | 하차 시각 (dropoff 시점) |
| **passenger\_count** | 탑승 승객 수 (※ 이 데이터에서는 전부 결측값) |
| **trip\_distance** | 이동 거리 (단위: 마일) |
| **ratecodeid** | 요금 코드 종류: 1: 표준 요금 2: JFK 3: 뉴어크 4: Negotiated fare 등 |
| **store\_and\_fwd\_flag** | 데이터 저장 후 전송 여부: Y: 일시 저장 후 전송 N: 실시간 전송 |
| **pulocationid** | 승차 지점의 지역 ID (택시존 기준) |
| **dolocationid** | 하차 지점의 지역 ID (택시존 기준) |
| **payment\_type** | 결제 방식: 1: 현금 2: 카드 3: 무료 승차 등 |
| **fare\_amount** | 기본 요금 |
| **extra** | 추가 요금 (야간 할증 등) |
| **mta\_tax** | MTA 부과세 (뉴욕 교통국 세금, 일반적으로 $0.50) |
| **tip\_amount** | 팁 금액 |
| **tolls\_amount** | 톨게이트 비용 (※ 이 데이터에서는 전부 결측값) |
| **improvement\_surcharge** | 인프라 개선 부과금 (일반적으로 $0.30) |
| **total\_amount** | 총 지불 금액 (팁 포함) |
| **congestion\_surcharge** | 혼잡 통행료 (일반적으로 $2.50, 데이터에 오류 있음) |

**Spark Monitoring and Logging 실습**

**Submit batch job run 클릭 후**

Script location:

**s3://us-east-1.elasticmapreduce/emr-containers/samples/wordcount/scripts/wordcount.py**

Script arguments:

**[“s3://emrserverless-workshop-891377038690/wordcount\_output\_consol\_logs/”]**

emrserverless-workshop-891377038690 버킷 안에 앞에서 만들어 놓은 경로 사용

Amazon S3 storage location URI :

**s3://emrserverless-workshop-891377038690/logs**

cloud9에서 다음 실행

(이부분은 앞에서 실행한것과 동일)

|  |
| --- |
| export JOB\_ROLE\_ARN=arn:aws:iam::891377038690:role/EMRServerlessS3RuntimeRole  export S3\_BUCKET=s3://emrserverless-workshop-891377038690  export APPLICATION\_ID=00frebt1ct62mm09 |

|  |
| --- |
| aws emr-serverless start-job-run --application-id ${APPLICATION\_ID} --execution-role-arn ${JOB\_ROLE\_ARN} --name "Spark-WordCount-CLI" --job-driver '{  "sparkSubmit": {  "entryPoint": "s3://us-east-1.elasticmapreduce/emr-containers/samples/wordcount/scripts/wordcount.py",  "entryPointArguments": [  "'"$S3\_BUCKET"'/wordcount\_output/"  ],  "sparkSubmitParameters": "--conf spark.executor.cores=1 --conf spark.executor.memory=4g --conf spark.driver.cores=1 --conf spark.driver.memory=4g --conf spark.executor.instances=1 --conf spark.hadoop.hive.metastore.client.factory.class=com.amazonaws.glue.catalog.metastore.AWSGlueDataCatalogHiveClientFactory"  }  }'\  --configuration-overrides '{  "monitoringConfiguration": {  "s3MonitoringConfiguration": {  "logUri": "'"$S3\_BUCKET"'/logs/"  }  }  }' |

텍스트, 스크린샷, 번호, 폰트이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

**[S3 Select를 사용한 쿼리]를 클릭하고 설정 값 변경 없이 바로 실행하고**

**[SQL 쿼리 실행] 버튼 클릭**

텍스트, 스크린샷, 소프트웨어, 웹 페이지이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

**스파크 UI**

옵션 2 - CLI에서 Spark 작업 제출의에서 실행한 아래 명령을 cloud9에서 실행한다

|  |
| --- |
| aws emr-serverless start-job-run --application-id ${APPLICATION\_ID} --execution-role-arn ${JOB\_ROLE\_ARN} --name "Spark-WordCount-CLI" --job-driver '{  "sparkSubmit": {  "entryPoint": "s3://us-east-1.elasticmapreduce/emr-containers/samples/wordcount/scripts/wordcount.py",  "entryPointArguments": [  "'"$S3\_BUCKET"'/wordcount\_output/"  ],  "sparkSubmitParameters": "--conf spark.executor.cores=1 --conf spark.executor.memory=4g --conf spark.driver.cores=1 --conf spark.driver.memory=4g --conf spark.executor.instances=1 --conf spark.hadoop.hive.metastore.client.factory.class=com.amazonaws.glue.catalog.metastore.AWSGlueDataCatalogHiveClientFactory"  }  }' |

EMR Studio -> Applications -> my-serverless-application에 가서 새로 생성된 Job이

Running 상태로 바뀔 때까지 기다렸다가 “ Spark UI(실행 중인 작업)”을 수행한다

수분내에 Success로 상태가 바뀌어서 볼 수가 없게 된다(실습불가)

AWS EMR Studio에서 Serverless Application으로 실행한 Spark Job이 너무 빠르게 종료되기 때문에 Spark UI를 확인할 수 없는 상황이다.

☞ 뒤에 **Spark 애플리케이션 모니터링** 실습에서 다시 자세히 학습한다

PySpark Python 소스를 수정하여 running time을 늘릴 수 있으나 비용이 발생하므로

다음 **CloudWatch metric** 실습으로 log를 확인한다

스택 생성시 주의사항

**텍스트, 스크린샷, 폰트이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.**

아래 EMR Studio -> Applications 에 있는

텍스트, 스크린샷, 폰트, 번호이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

ApplicationID를 복사해서 반드시 입력해준다(미 입력 시 스택 생성 오류 발생)

**텍스트, 스크린샷, 폰트, 번호이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.**

CloudWatch -> Dashboards -> emr-serverless-dashboard-00frebt1ct62mm09 확인

**텍스트, 스크린샷, 도표, 그래프이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.**

**하이브 모니터링 및 로깅**

**Application logs and metrics ->**

Upload logs to your Amazon S3 bucket:

**s3://emrserverless-workshop-891377038690/hive-logs/**

추가로 설정해준다 (Workshop에는 엑박으로 잘 안 보임)

**하이브 테즈 UI (실습 불가)**

마찬가지로 Hive Job이 너무 빠르게 종료되기 때문에 Hive Tez UI를 확인할 수 없다

**Apache Spark를 위한 Amazon Redshift 통합**

현재 EMR Release version이 emr-7.8.0 이므로

**EMR 릴리스 버전 6.10.0 이상**으로 실습 진행

텍스트, 스크린샷, 폰트, 번호이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

압축 파일을 풀어서

Amazon S3 ->버킷->emrserverless-workshop-891377038690/

아래에 업로드한다 (주의 상위 spark-redshift 폴더가 아닌 하위 폴더 2개를 따로 두번 선택한다음 [업로드] 버튼을 누른다)

업로드 결과 아래와 같은 경로가 된다

emrserverless-workshop-891377038690/script/

emrserverless-workshop-891377038690/input/

Amazon S3 ->버킷->emrserverless-workshop-891377038690/ 아래에

redshift-spark-demo-temp-dir 폴더를 하나 만들어 놓는다

EMR Studio ->Applications 에서 my-serverless-application을 선택한 다음  
Actions에서 Edit을 클릭한다

텍스트, 스크린샷, 폰트, 번호이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

Network connections -> Virtual private cloud (VPC) 만 수정 (다른 옵션은 그대로 둔다)  
CloudFormation의 출력 탭에서 VpcId의 값을 확인하여 동일한 걸로 선택한다

Subnet은 VPC로가서 생성된 VPC를 클릭하여 하위 서브넷을 확인한다

우측 IGW 를 선택하면 public subnet 3개를 확인할 수 있다 이 세 개를 EMS studio에서

선택해준다

텍스트, 스크린샷, 폰트, 번호이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

텍스트, 스크린샷, 폰트, 번호이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

Redshift 데이터베이스 이름은 변경 불가 : test로 그대로 둔다

수동으로 비번 변경 : Password1234 (반드시 대/소문자와 숫자 포함)

쿼리 편집기로 이동 후 초기 Configure Account는 생략 가능

CloudFormation의 출력 탭에서와 EMR studio, AWS 계정ID를 복사하여 사용

cloud9에서 다음을 실행한다

|  |
| --- |
| export JOB\_ROLE\_ARN=arn:aws:iam::891377038690:role/EMRServerlessS3RuntimeRole  export S3\_BUCKET=s3://emrserverless-workshop-891377038690  export APPLICATION\_ID=00frebt1ct62mm09  export ACCOUNT\_ID=891377038690 |

cloud9에서 다음을 실행한다

|  |
| --- |
| aws emr-serverless start-job-run --application-id ${APPLICATION\_ID} --region us-east-1 --execution-role-arn ${JOB\_ROLE\_ARN} --name "Redshift-Redshift-Job" --job-driver '{  "sparkSubmit": {  "entryPoint": "'"$S3\_BUCKET"'/script/load\_to\_redshift.py",  "entryPointArguments": ["redshift","'"$ACCOUNT\_ID"'"]  }  }' --configuration-overrides '{  "monitoringConfiguration": {  "s3MonitoringConfiguration": {  "logUri": "'"${S3\_BUCKET}"'/logs/"  }  }  }' |

실행 후 터미널에 출력된 job-run-id를 사용하여 아래 명령으로 Job 실행 확인 가능

aws emr-serverless get-job-run --application-id ${APPLICATION\_ID} --job-run-id 00frg4idk2cev00b --region us-east-1

EMR Studio ->Applications -> my-serverless-application 에서도 확인 가능

텍스트, 폰트, 라인, 스크린샷이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

EMR Studio에서 Spark 작업 제출 부분에서

Script location 입력을 아래와 같이 입력한다

s3://emrserverless-workshop-891377038690/script/load\_to\_redshift.py

Script arguments:

["S3", "891377038690"]

EMR Studio ->Applications -> my-serverless-application 에서 Job 실행 확인

텍스트, 폰트, 라인, 번호이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

Redshift 쿼리 에디터에서는 테이블을 선택하고 마우스 우측 버튼을 클릭하면

Select table 메뉴가 보인다 여기를 클릭하면 자동으로 쿼리 스크립트가 생성된다

텍스트, 스크린샷, 폰트, 소프트웨어이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다. 텍스트, 스크린샷, 폰트, 소프트웨어이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

**EMR 서버리스 애플리케이션 생성**

이 실습을 수행하려면 자신의 계정에서 다른 사용자로 로그인 해야 한다

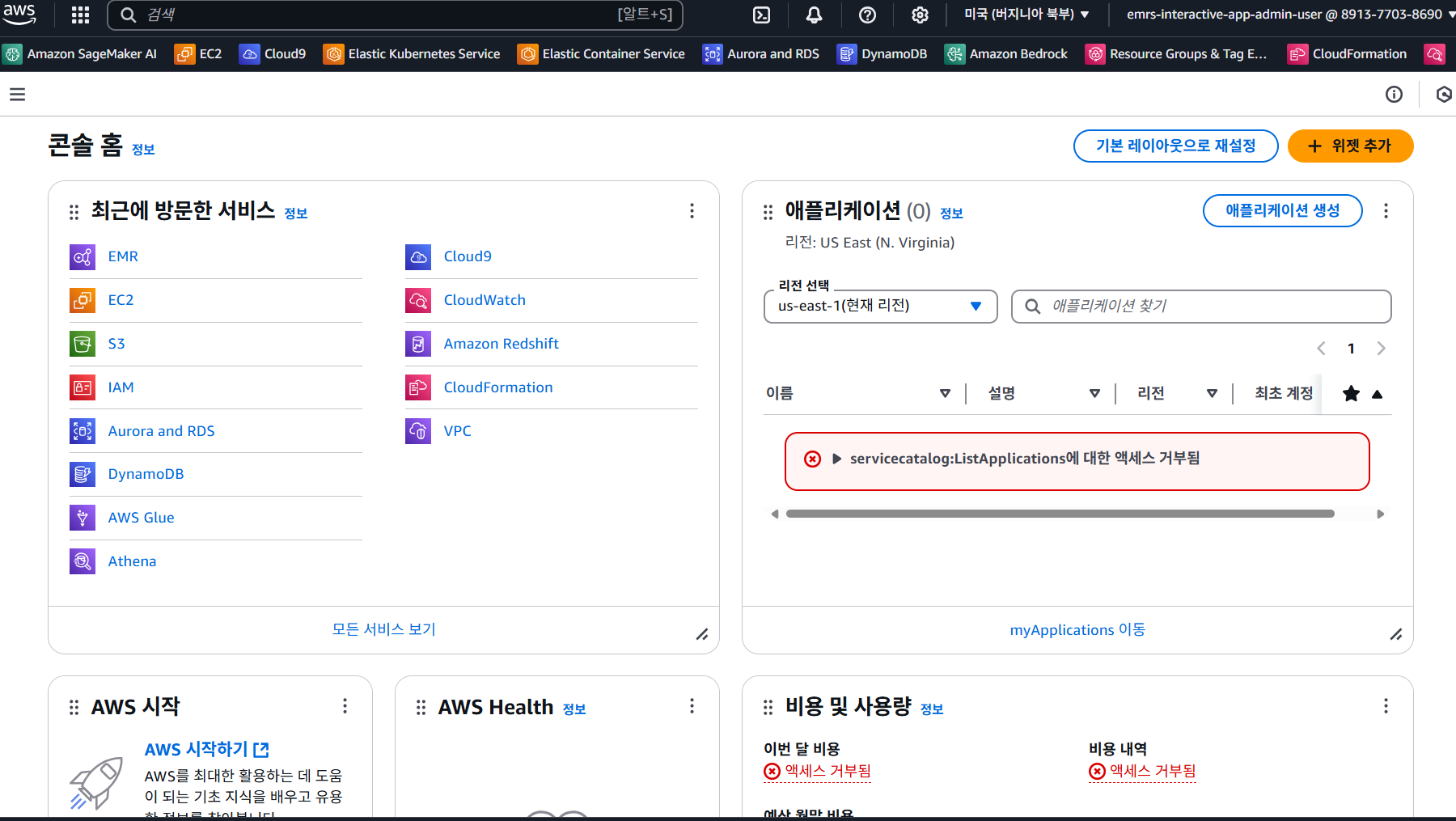
|  |
| --- |
| 사용자 ID : emrs-interactive-app-admin-user  암호: myP@ssW0rd |

Edge나 다른 브라우저 창을 새로 열고 자신의 AWS 계정 ID를 복사하여 접속한다

**https://<계정ID>.signin.aws.amazon.com/console**

예) <https://890377048610.signin.aws.amazon.com/console>

텍스트, 스크린샷, 폰트, 번호이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다. 

AWS 콘솔에 로그인되면 리전을 미국(버지니아 북부)로 변경해준다

Create application 🡪

Application settings -> Name:

my-serverless-interactive-application

Interactive endpoint 에서

Enable endpoint for EMR studio 을 선택 체크한다

**Network connections**

텍스트, 스크린샷, 소프트웨어, 번호이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

**studio-1 편집 -> Workspace 스토리지:**

s3://emrserverless-interactive-blog-891377038690-us-east-1

EMR Studio ->Workspace에서 새 팝업으로 새 창이 안 열리면 화면 우측 상단에서

사이트 설정을 클릭하고 Pop-ups and redirects을 허용(Allow)으로 변경해준다

- Edge 에서 : 자물쇠 아이콘 🡪 이 사이트에 대한 사용권한 클릭

텍스트, 스크린샷, 폰트, 번호이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

텍스트, 스크린샷, 번호, 폰트이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

- Chrome 에서 : Site settings

텍스트, 스크린샷, 번호, 폰트이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

텍스트, 스크린샷, 번호, 폰트이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

**Spark 애플리케이션 모니터링** 설명

**[1] Apache Spark Web UI**의 **Jobs 페이지**

텍스트, 스크린샷, 라인, 그래프이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

**Apache Spark Web UI**의 **Jobs → Timeline** 뷰입니다.  
Spark 애플리케이션 실행 중에 **Executor가 추가된 시점과 Job 상태**를 시각적으로 준다

**✅ 화면 구성 설명**

**🧾 상단 정보**

| **항목** | **설명** |
| --- | --- |
| **Spark 버전** | 3.5.4-amzn-0 (Amazon EMR에서 실행된 Spark 버전) |
| **User** | hadoop — Spark 애플리케이션을 실행한 사용자 |
| **Total Uptime** | 1.1분 — 실행된 지 1분 6초 정도 경과 |
| **Scheduling Mode** | FIFO — 작업 스케줄링 방식 (First In First Out) |

**📊 하단 그래프 해석**

**1. Executors 영역 (위쪽)**

Spark가 작업을 수행하기 위해 사용한 **Executor들의 생성 이벤트**가 나옵니다.

| **시점** | **이벤트** |
| --- | --- |
| 4월 4일 21:34:48 | Driver 추가 (Executor driver added) |
| 21:34:54~55 | Executor 1, 2, 3 순차적으로 추가됨 |

✅ Executor는 Spark에서 실제 데이터를 처리하는 병렬 작업 단위입니다.

**2. Jobs 영역 (아래쪽)**

* 작업(Job)이 실행된 타임라인을 보여줍니다.
* 현재는 Job이 성공했는지(Succeeded), 실패했는지(Failed), 실행 중인지(Running) 표시 없음 → 아마 Job이 아직 시작되지 않았거나 너무 짧았던 것일 수 있음

🟢 = Running, 🔴 = Failed, 🟣 = Succeeded로 표시됩니다 (지금은 아무 색도 없음)

**💡 해석 요약**

* 드라이버가 먼저 시작됨 (Driver: 명령 분배 역할)
* 곧이어 Executor 1, 2, 3이 거의 동시에 할당됨 → 병렬 작업 준비 완료
* Job에 대한 시각화는 이 시점에서는 표시되지 않았지만, 곧 실행될 가능성 있음

**🧠 이 화면이 중요한 이유**

| **이유** | **설명** |
| --- | --- |
| 병렬 처리 확인 | Executor가 얼마나 빠르게 배치되었는지 확인 가능 |
| 자원 할당 분석 | 필요한 만큼 Executor가 잘 생성되었는지 확인 |
| 성능 이슈 분석 | Executor가 늦게 생성되거나 너무 적으면 병목 가능성 |

**Spark 개념 정리**

**🚀 1. 드라이버(Driver) vs 실행자(Executor)**

| **구성 요소** | **설명** | **비유** |
| --- | --- | --- |
| **Driver** | Spark 애플리케이션의 **진행을 조정하는 컨트롤 타워**입니다. 사용자 코드가 실행되며, Executor에게 작업을 분배합니다. | 👩‍💼 매니저 |
| **Executor** | 실질적으로 데이터를 처리하는 작업자입니다. 드라이버가 할당한 태스크를 받아서 병렬로 처리합니다. | 🧑‍🔧 직원들 |

**흐름:**

|  |
| --- |
| 사용자 코드 (SparkSession)  ↓  Driver가 실행됨  ↓  Cluster에 Executor 여러 개 배포  ↓  Driver가 Task를 나눠서 Executor에게 분산 전송  ↓  Executor들이 병렬 처리 |

**🧩 2. 태스크(Task) 분산 구조**

Spark는 **RDD(Resilient Distributed Dataset)** 또는 **DataFrame**을 여러 파티션으로 나눕니다.  
각 파티션은 Executor에게 전달되어 **하나의 Task**로 처리됩니다.

| **개념** | **설명** |
| --- | --- |
| **Stage** | Job을 DAG로 나누었을 때 동일한 셔플 이전/이후 연산 묶음 |
| **Task** | 각 Stage가 파티션 단위로 나뉘어 실행되는 단위 |
| **Executor** | 여러 Task를 병렬로 실행 |

**예시**

* 데이터 1TB → 200 파티션
* 각 Executor가 4개씩 처리 → 50개의 Executor 생성 가능

**♻️ 3. GC Time (Garbage Collection Time)**

* Spark는 **JVM 기반**이기 때문에, 메모리를 자동으로 관리합니다.
* 오래 실행되거나, 메모리가 부족하면 \*\*GC(Garbage Collector)\*\*가 자주 돌게 됩니다.
* GC Time이 길수록 실제 작업 시간이 줄어들고 성능이 나빠집니다.

| **상황** | **의미** |
| --- | --- |
| GC Time 0~1초 | 정상 |
| GC Time > 10초 | 성능 저하 원인 가능성 있음 |
| GC Time이 전체 실행 시간의 20% 이상 | 메모리 튜닝 필요 |

**🔄 4. Shuffle 개념**

**Shuffle**은 Spark에서 **데이터를 Executor 간에 재분배하는 과정**입니다.  
예를 들어 groupBy, join, reduceByKey 등의 연산은 파티션을 다시 섞어야 하므로 셔플이 발생합니다.

| **항목** | **설명** |
| --- | --- |
| **Shuffle Read** | 다른 Executor로부터 받은 데이터 양 |
| **Shuffle Write** | 다른 Executor에게 보낸 데이터 양 |

**특징:**

* **셔플이 많으면 성능 저하** (디스크 I/O + 네트워크 비용)
* 셔플이 너무 많으면 **스테이지 간 병목**이 생김

**✨ 요약 그림**

[Spark Driver]

|

├── Stage 1

│ ├── Task 1 (→ Executor 1)

│ ├── Task 2 (→ Executor 2)

│ └── ...

|

└── Shuffle 발생 (예: groupBy)

↓

Stage 2

├── Task 1 (→ Executor 3)

├── Task 2 (→ Executor 4)

**🛠️ Spark 튜닝이 필요한 상황은?**

* GC Time이 너무 높다 → 메모리 부족 or 데이터 skew (데이터 쏠림 현상 ->

데이터가 일부 키나 파티션에 몰려서, 일부 Executor만 과도한 작업을 하게 되는 현상)

* Shuffle이 너무 많다 → 잘못된 파티셔닝 or groupBy 쏠림
* Executor 수가 너무 적거나 많다 → 리소스 낭비 or 병렬성 부족

**“Getting-started-with-emr-serverless.ipynb” 파이썬 코드 모두 실행 후 Jobs 페이지** 텍스트, 스크린샷, 번호, 소프트웨어이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

**✅ 전체 개요**

* 총 **17개의 Job**이 실행되었고, 모두 \*\*성공(Succeeded)\*\*했습니다.
* 작업은 대부분 toPandas, runJob, showString, load, install\_pypi\_package 등 Spark에서 흔히 사용하는 연산입니다.
* 시간은 2025/04/04 22:18:57부터 약 2분 사이에 실행되었습니다.

**🧩 Job 세부 분석**

**🔹 Job 0**

* **Statement 2**: install\_pypi\_package
* **의미**: PyPi 패키지 설치 명령 실행. 예: pip install somepackage
* **Duration**: 7초
* **결과**: 3개의 태스크 모두 성공 (3/3)

**🔹 Job 1**

* **Statement 3**: load
* **의미**: 데이터를 로드하는 Spark 작업 (예: spark.read...)
* **Duration**: 1초
* **결과**: 정상 (1/1)

**🔹 Job 2~3**

* **Statement 4**: showString
* **의미**: df.show() 또는 출력용 .display() 같은 코드
* **Duration**: 3초 (Job 2), 0.2초 (Job 3)
* **결과**: 첫 시도에 전체 로딩, 두 번째는 캐시된 데이터만 출력했을 가능성

**🔹 Job 4~7**

* **Statement 5**: runJob at PythonRDD.scala:191
* **의미**: 파이썬 기반 RDD 작업 (예: map, filter, collect, count)
* **특징**:
  + 여러 job으로 나뉘어 실행된 것 보아 **여러 번 반복 실행**, 또는 **여러 파티션을 점진적으로 처리**한 것으로 보임
  + 실행 시간: 0.4~2초
  + 처리 데이터량: 4개, 13개, 20개, 100개 등 다양

**🔹 Job 8~9**

* **Statement 6**: showString
* **의미**: 다시 .show() 출력 수행
* **특징**: 전체 12개 태스크 처리 (데이터 12개?), duration 1초

**🔹 Job 10~12**

* **Statement 7**: runJob
* **의미**: RDD 또는 DataFrame 변환 작업
* **성능**: 0.8초 이하로 빠름

**🔹 Job 13~16**

* **Statement 8**: toPandas
* **의미**: df.toPandas() 호출 → 전체 데이터를 드라이버 메모리로 가져오기
* **특징**:
  + 여러 job으로 나뉨 (Spark가 내부적으로 병렬 fetch 수행)
  + 각 job은 평균 65~100ms로 매우 빠름
  + 일부 태스크/스테이지가 스킵됨 → 중복 실행이 있었거나 캐시된 상태

**🧠 총평: 로그를 통해 알 수 있는 것**

| **관찰 항목** | **의미** |
| --- | --- |
| ✅ 모든 Job 성공 | 코드나 환경 문제 없이 정상 작동 중 |
| 🔁 같은 Statement의 반복 실행 | 여러 번 .show(), .toPandas() 등 확인 작업 수행 |
| ⏱ 평균 처리 시간 빠름 | 작업량이 적거나 클러스터 리소스가 충분함 |
| 🧮 대부분 12개 태스크 | 데이터 샘플 수 또는 RDD 파티션 수 추정 가능 |

**🔍 다음 단계에서 할 수 있는 것**

* 만약 **Job이 느리거나 실패**했다면, stderr 또는 driver logs를 열어야 분석 가능
* 현재처럼 toPandas()가 여러 번 수행되면 **메모리 부담**이 크므로 **필요할 때만 호출**하는 것이 좋음
* show()는 기본 20개까지 출력하므로 데이터의 전체 크기를 모르고 그냥 봤다면, .count() 또는 .summary() 활용 추천

**[2] Apache Spark Web UI의 Executors 페이지**

텍스트, 스크린샷, 번호, 폰트이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

Spark 애플리케이션이 실행되었을 때 사용된 **Executor(실행 노드)들의 상태와 리소스 사용량을 실시간으로 보여주는 대시보드**이다.

**✅ Executors Summary에서 표시되는 항목 설명**

Active(4) : 현재 정상 작동 중인 Executor 수, 현재 4개의 Executor가 Spark 클러스터에서 정상 작동 중

Dead(0) – 비정상 종료되었거나 작업 완료 후 종료된 Executor 수

💡 언제 중요할까?

* Dead 수가 많다면 → 작업 실패 원인 분석 필요
* Executor가 모두 Dead, Active = 0이라면 → 작업 완료 or 클러스터 자원 부족

**✅ Executors Executors에서 표시되는 항목 설명**

Spark 실행 UI의 Executors 상세 목록 테이블입니다.  
각 Executor 인스턴스(= Spark 작업 노드)의 상태, 리소스, 작업 처리 정보 등을 보여주는 핵심 진단 자료이다

**✅ 테이블 요약 해석 (현재 상태 기준)**

| **항목** | **의미** | **해석** |
| --- | --- | --- |
| **Executor ID** | 실행 ID (driver, 1, 2, 3) | Driver는 메인 컨트롤러, 1~3은 작업자 |
| **Address** | 할당된 IPv6 주소 + 포트 | 각 Executor가 배치된 노드 주소 |
| **Status** | 현재 상태 (Active) | 모두 정상적으로 동작 중 👍 |
| **RDD Blocks** | 메모리에 캐시된 블록 수 | 0 → 캐싱된 데이터 없음 |
| **Storage Memory** | 사용 메모리 / 총 메모리 | 모두 0.0 B / 7.9 GiB → 아직 메모리 사용 안 됨 |
| **Disk Used** | 임시 저장한 디스크 공간 | 0.0 B → 디스크 사용 없음 |
| **Cores** | Executor가 사용하는 CPU 코어 수 | 각 4개 → 총 12개 |
| **Active Tasks** | 현재 실행 중인 태스크 수 | 0 → 아직 실행 중인 작업 없음 |
| **Failed / Complete / Total Tasks** | 태스크 상태 | 모두 0 → 작업 수행 안 함 |
| **Task Time (GC Time)** | 작업 시간 / GC 시간 | 드라이버만 15분 대기, GC 거의 없음 |
| **Input / Shuffle Read / Write** | 데이터 입출력량 | 모두 0.0 B → 데이터 읽거나 쓴 기록 없음 |
| **Logs** | stderr / stdout 로그 링크 | 클릭 시 해당 Executor의 로그 확인 가능 |
| **Thread Dump / Heap Histogram** | JVM 내부 상태 디버깅 도구 | GC, 스레드, 힙 메모리 분석용 |
| **Add Time** | Executor 시작 시각 | 모두 2025-04-05 06:34:54 |
| **Remove Time** | 제거 시각 | 아직 없음 (-) → 정상 동작 중 |

**파이썬 코드 모두 실행 후 Executors 페이지**

**텍스트, 스크린샷, 번호, 폰트이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.**

**✅ Executors 표 내용 요약**

| **항목** | **값** | **의미** |
| --- | --- | --- |
| **Active Executors** | 4 | 현재 4개의 executor가 살아있고 작업 중 |
| **RDD Blocks** | 0 | 메모리에 캐시된 RDD 없음 (persist/cache 안 씀) |
| **Storage Memory** | 4.3 MiB / 31.8 GiB | 전체 메모리 중 일부만 사용됨 |
| **Disk Used** | 0.0 B | 디스크 스필 없음 (메모리 내 작업 완료) |
| **Cores** | 12 | 총 12개의 코어 사용 (3개의 executor x 4코어 + 드라이버는 0) |
| **Total Tasks** | 259 | 모든 executor에서 수행한 태스크 총합 |
| **Task Time (GC Time)** | 58분 (0.5초) | 전체 태스크 수행 시간 중 GC는 매우 적음 (매우 양호) |
| **Input / Shuffle** | 55.4 MiB 입력 / 7.8 KiB 읽기 / 6.4 KiB 쓰기 | 소규모 데이터 처리 |

**🧩 각 Executor 상세 분석**

| **Executor ID** | **역할** | **Storage Memory** | **Cores** | **Total Tasks** | **Task Time** | **Input** | **Shuffle** |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **driver** | 드라이버 (명령 실행 중심) | 1.5 MiB / 8.2 GiB | 0 | 0 | 57분 대기 | 0 B | 0 B |
| **1** | Executor | 893.1 KiB / 7.9 GiB | 4 | 83 | 24초 | 27.7 MiB | 5.3 KiB / 3.9 KiB |
| **2** | Executor | 1022 KiB / 7.9 GiB | 4 | 87 | 19초 | 0 B | 1.4 KiB / 0 B |
| **3** | Executor | 912.4 KiB / 7.9 GiB | 4 | 89 | 24초 | 27.7 MiB | 1.2 KiB / 2.5 KiB |

**해석:**

* 각 executor는 4코어씩 할당됨
* 작업량은 거의 균등하게 배분됨 (83~89 tasks)
* 입출력(IO)은 매우 적고 메모리 중심 처리
* GC 시간도 매우 짧고 리소스 상태도 양호

**🧠 분석 포인트 및 추천**

**🔸 성능적으로 볼 때**

* 작업 병렬화 잘 됨 (3개의 executor가 거의 균등 작업)
* GC 부하 없음 → 메모리 적절 사용
* 디스크 스필 없음 → memoryOnly 전략 적합
* 259개의 태스크 처리에 약 1분 → 전체 작업 효율적

**🔸 튜닝 필요 없음**

* 현재는 executor 수, 코어 수, 메모리 모두 적절
* 만약 데이터 양이 많아진다면 이후에는 Storage Memory, Shuffle Spill, Disk Used 등을 주의 깊게 봐야 함

**🔧 추가로 확인 가능한 것**

| **항목** | **방법** |
| --- | --- |
| 태스크별 실행 시간 / 실패 여부 | Spark UI의 "Stages", "Tasks" 탭 확인 |
| Executor 로그 | 각 executor 옆의 stderr, stdout 클릭 |
| Thread Dump | JVM 상태, 병목 진단 가능 |
| Heap Histogram | 메모리 객체 분석 가능 (튜닝 시 활용) |

**🔎 1. 리소스 사용 분석**

| **항목** | **값** | **해석** |
| --- | --- | --- |
| Executors | 4 (3 Executor + 1 Driver) | 병렬 처리 구조 적절 |
| Cores | 12 | 4코어 × 3 Executor |
| Storage Memory 사용량 | 4.3 MiB / 31.8 GiB | 전체 메모리 대비 0.01%도 안 씀 → 매우 여유 |
| Disk Used | 0.0 B | 디스크 스필 없음 → 메모리 내에서 모든 처리 완료 |

✅ **결론**: 메모리, CPU 모두 매우 여유로운 상태. 자원 낭비 없이 깔끔하게 처리됨.

**🔎 2. 작업 처리 상태 분석**

| **항목** | **값** | **해석** |
| --- | --- | --- |
| 총 태스크 수 | 259 | 전체 Job의 task 수합 |
| 실패 태스크 | 0 | 작업 안정성 매우 높음 |
| GC 시간 | 전체 0.5초 / 58분 실행 | 거의 없을 정도로 안정적 |

✅ **결론**: 태스크 분산 및 실행 안정성 매우 우수. Java/Python GC로 인한 병목 없음.

**🔎 3. Executor별 처리 상태**

| **Executor ID** | **Tasks** | **Task Time** | **Input** | **Shuffle Read/Write** |
| --- | --- | --- | --- | --- |
| 1 | 83 | 24초 | 27.7 MiB | 5.3 KiB / 3.9 KiB |
| 2 | 87 | 19초 | 0 B | 1.4 KiB / 0 B |
| 3 | 89 | 24초 | 27.7 MiB | 1.2 KiB / 2.5 KiB |

**🧠 해석:**

* **Executor 2**는 입력 데이터가 0 → **캐시된 데이터** 또는 **join 없이 처리된 태스크**
* **Executor 1과 3**만 실제 데이터 읽음 → 분산 처리 비율이 100% 균등은 아님
* Shuffle 입출력량은 매우 적음 → Join, GroupBy, Aggregation 등의 복잡한 연산이 거의 없음

✅ **결론**: 데이터 입출력이 적고, Shuffle 비용도 거의 없는 lightweight 작업. map, filter, toPandas, show() 중심 작업으로 추정됨.

**🔎 4. 의심 혹은 주의할 점**

| **항목** | **설명** |
| --- | --- |
| toPandas() 반복 호출 | 드라이버 메모리에 데이터 복사 → 대규모 데이터에서는 OOM 위험 |
| RDD Blocks = 0 | cache(), persist()가 전혀 사용되지 않음 → 동일 데이터 여러 번 읽으면 비효율 발생 가능 |
| **Driver에 태스크 없음** | 분석 용도로는 문제가 없지만, 복잡한 연산의 경우 드라이버의 처리도 점검 필요 |

**✅ 종합 결론**

**전체 작업은 경량 데이터셋을 대상으로, 분산 환경에서 매우 안정적으로 수행되었으며, 리소스 활용률이 낮고 성능 병목도 없음.**

**그러나**:

* toPandas()를 반복적으로 호출하고 있어, **데이터 크기가 증가하면 위험 요소**로 작용 가능
* persist() 등을 통해 반복 사용하는 데이터는 메모리에 올려두는 전략도 고려 필요

**💡 추천 조치**

| **상황** | **조치** |
| --- | --- |
| 동일한 DataFrame 반복 사용 시 | .cache() 또는 .persist() 사용 |
| 대규모 데이터 처리 예정 | .toPandas()는 피하고 .write.format("csv").save(...) 등 분산 방식 사용 |
| 태스크 처리 시간 세부 분석 원할 때 | Stages → Tasks → Gantt View 확인 추천 |

**[3] Stages  페이지 : 노트북에서 파이썬 코드 모두 실행 후에 확인 가능**

텍스트, 스크린샷, 번호, 라인이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

AWS EMR Serverless에서 실행된 Spark 애플리케이션의 "Stages" 탭 (완료된 작업)에 직접 연결되는 대시보드 URL입니다.

Spark에서 "Stage"는 작업(Job)이 실행되는 **논리적 처리 단위**이며, **작업이 어떻게 분할되어 병렬로 실행되었는지** 보여주는 중요한 분석 포인트입니다.

**🧩 Stage 탭에서 볼 수 있는 주요 정보 설명**

| **항목** | **의미** | **분석 포인트** |
| --- | --- | --- |
| **Stage ID** | 스테이지 고유 번호 | 각 Job 안에서의 작업 단위 |
| **Description** | 작업 설명 | mapPartitions, filter, shuffle, sort 등 Spark 연산자 이름 |
| **Submitted** | 시작 시각 | 병렬 처리 타이밍 확인 |
| **Duration** | 실행 시간 | 병목 구간 진단 가능 |
| **Tasks: Succeeded/Total** | 성공/총 태스크 수 | 실패 태스크 발생 여부 |
| **Input Size / Records** | 이 스테이지가 처리한 데이터 양 | DataFrame 연산 중 I/O 규모 확인 |
| **Shuffle Read/Write** | 다른 스테이지와의 데이터 교환량 | **Join, GroupBy, Sort** 연산 시 필수 분석 지표 |
| **GC Time** | 가비지 컬렉션 시간 | 메모리 병목 여부 판단 기준 |
| **Skew** | Task 간 작업 편차 | 특정 태스크가 오래 걸렸는지 여부 판단 |

**📊 예시로 보는 해석 (표는 예시 구조)**

| **Stage ID** | **Description** | **Duration** | **Tasks** | **Input** | **Shuffle Read/Write** | **GC Time** |
| --- | --- | --- | --- | --- | --- | --- |
| 13 | mapPartitions at toPandas | 400ms | 12/12 | 55 MiB | 0 / 0 | 0ms |
| 14 | collect at toPandas | 200ms | 1/1 | 0 B | 55 MiB / 0 B | 0ms |

🔎 **해석**:

* Stage 13: 전체 데이터를 executor들이 병렬로 읽음 (mapPartitions)
* Stage 14: 데이터를 드라이버로 가져오는 단계 (collect), toPandas()의 마지막 스텝
* Shuffle read가 Stage 14에 몰림 → Spark가 데이터를 드라이버로 "끌어오는" 상황
* 모든 Task는 12개로 균등함 → 파티션이 12개로 나뉜 구조
* 실행 시간 빠르고 GC 없음 → 병목 없음

**🧠 주요 분석 팁**

| **상황** | **확인 포인트** |
| --- | --- |
| 작업이 느리다 | Duration, Task Time, Skew 값을 확인 |
| OOM 오류 발생 | Input Size, Shuffle Size, GC Time을 비교 분석 |
| 특정 태스크만 느리다 | Gantt Chart에서 **Task 실행 시간 분포** 확인 가능 |
| 무의미한 반복 스테이지 존재 | 같은 description이 여러 번 나오는지 확인 → 캐시 또는 코드 개선 필요 |

**🛠️ 추가 분석 가능 항목**

Spark UI의 각 Stage를 클릭하면 다음 정보도 확인할 수 있습니다:

* **Task별 Gantt 차트**: 태스크 시작/종료 시간 시각화
* **Summary Metrics for Tasks**:
  + 평균/최대 실행 시간
  + Input/Output Size 분포
* **Executor Breakdown**:
  + 각 executor가 처리한 task 수, 속도, shuffle 사용량 등

**✅ 결론**

이 stages/#completed 화면은 Spark 작업을 **성능 및 효율성 관점에서 진단하는 핵심 도구**입니다.  
특히 다음과 같은 질문에 답할 수 있게 해줍니다:

* 데이터가 제대로 분산되었는가?
* 느린 태스크는 어디서 발생했는가?
* 병목 구간은 어느 Stage인가?
* 리소스 과다 사용은 없었는가?

.

**✅ 1. Stages for All Jobs 전체 요약**

* **Completed Stages: 18개** → 실행되어 성공적으로 완료된 작업 단계
* **Skipped Stages: 10개** → 이전 실행 결과가 캐시되어 재실행이 생략된 단계 (스파크 최적화 효과)

**🧩 2. Completed Stages 분석**

여기선 중요한 몇 개만 대표적으로 요약해드립니다:

**🔹 Stage 20**

* **작업**: toPandas
* **태스크**: 12개 (병렬 실행)
* **Input**: 24.3 MiB
* **Shuffle Write**: 1.4 KiB
* ✅ → toPandas() 호출 시 실제 데이터 병렬로 수집한 작업

**🔹 Stage 15**

* **작업**: showString
* **태스크**: 12/12
* **Input**: 24.3 MiB
* **Output**: 1416 B
* ✅ → .show() 호출로 추정됨, 출력은 작지만 내부적으로는 전체 데이터를 로딩

**🔹 Stage 12**

* **작업**: runJob
* **태스크**: 100/100
* **Input**: 515 B
* ✅ → 매우 가벼운 작업 (예: 작은 RDD 연산)

**🔹 Stage 5**

* **작업**: toJavaRDD
* **태스크**: 12/12
* **Input**: 3.4 MiB
* **Output**: 1140 B
* ✅ → df.rdd 또는 PySpark-RDD 변환 추정

**🔹 Stage 0**

* **작업**: install\_pypi\_package
* **태스크**: 3/3
* ✅ → 외부 PyPI 패키지 설치 완료됨 (예: !pip install ...)

**⚠️ 3. Skipped Stages 분석 (총 10개)**

| **Stage ID** | **Description** | **추정 원인** |
| --- | --- | --- |
| 26, 25, 23, 21 | toPandas | 이전 실행 결과가 캐시되어 재실행 생략 |
| 13, 11, 9, 7 | toJavaRDD | 동일한 변환 재사용으로 인한 skip |
| 16, 3 | showString | 출력 중복 요청 생략됨 (.show() 반복 호출) |

🧠 **Spark의 DAG(Directed Acyclic Graph) 최적화** 덕분에, 동일한 변환 결과가 중복 실행되지 않고 **스킵 처리**된 것입니다.

**📊 성능 측면 핵심 요약**

| **지표** | **상태** | **해석** |
| --- | --- | --- |
| Input 처리량 | ~24 MiB 수준 | 중간 크기 데이터, 병렬성 적절 |
| 태스크 분할 | 대부분 12개 | 클러스터 파티셔닝 정상 |
| 실패 | 없음 | 안정적 실행 |
| Shuffle 발생 | 매우 적음 (1.4 KiB 수준) | Join, GroupBy 같은 복잡 연산 없음 |
| Cache 활용 | 일부 toPandas, toJavaRDD 결과가 캐시됨 | 성능 최적화 발생 |

**✅ 최종 정리**

전체 작업은 주로 toPandas, show(), RDD 변환, runJob 등으로 구성된 가볍고 안정적인 PySpark 실행 흐름입니다.

* 실행된 18개 스테이지는 대부분 **데이터 탐색용 (EDA)** 중심
* 중복된 변환은 Spark가 캐시하여 10개 스테이지는 **스킵 처리**됨 (효율적!)
* 성능 병목, 리소스 과부하, 실패 없음 → 매우 깔끔한 실행 결과

[4] **SQL / DataFrame 페이지 : 노트북에서 파이썬 코드 실행 후 확인**

텍스트, 스크린샷, 라인, 폰트이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

**PySpark 또는 Spark SQL로 실행한 쿼리의 이력, 상태, 결과를 확인할 수 있는 페이지**

**✅ SQL/DataFrame 쿼리 실행 이력**

| **Query ID** | **설명 (Statement)** | **제출 시각** | **소요 시간** | **연결된 Job ID** |
| --- | --- | --- | --- | --- |
| **4** | Statement 8 | 2025/04/04 22:21:13 | 2초 | [13][14][15][16] |
| **3** | Statement 7 | 2025/04/04 22:20:12 | 1초 | (Job ID 없음 표기 → 내부 작업만 수행했을 가능성) |
| **2** | Statement 6 | 2025/04/04 22:20:10 | 2초 | [9][10] |
| **1** | Statement 6 (아마도 중복 실행) | 2025/04/04 22:20:09 | 9ms | - |
| **0** | Statement 4 | 2025/04/04 22:19:33 | 4초 | [2][3] |

**🔍 주요 쿼리 상세 분석**

**🔹 Query ID 4: Statement 8 (toPandas)**

* ⏱ **Duration**: 2초
* 📦 **Job 연결**: 4개 ([13][14][15][16])
* 🔍 **해석**:
  + PySpark DataFrame을 .toPandas()로 변환한 명령
  + 이 작업은 보통:
    1. Executor에서 데이터를 모으고
    2. Driver로 수집한 후
    3. Pandas DataFrame으로 변환
  + 여러 Job으로 나뉜 이유는 Spark 내부에서 **RDD 변환 + 수집(collect) + 변환(toPandas)** 단계가 분리됨

**🔹 Query ID 3: Statement 7**

* ⏱ **Duration**: 1초
* 🧩 Job ID 없음 → 메타데이터 작업 가능성 (예: .count(), .schema 확인 등)
* 가능성:
  + Lazy evaluation이므로 이 쿼리는 transformation만 수행되었을 수 있음
  + 아니면 Spark이 캐시된 결과를 사용했을 수 있음

**🔹 Query ID 2: Statement 6 (show())**

* ⏱ **Duration**: 2초
* 📦 **Job ID**: [9], [10]
* 🔍 해석:
  + df.show() 같은 명령으로, 실제 실행에서 데이터를 출력하는 Job 발생
  + 2개의 Job으로 분리된 이유는:
    - 첫 Job: Executor에서 데이터를 읽음
    - 둘째 Job: 데이터를 포맷팅하여 Driver에 출력

**🔹 Query ID 1: Statement 6 (중복 실행)**

* ⏱ **Duration**: 9ms (매우 빠름)
* 📦 Job 없음
* 🧠 해석: 이전 실행 결과가 **캐시되어 재사용**됨 → **Spark의 최적화 작동**

**🔹 Query ID 0: Statement 4 (show())**

* ⏱ **Duration**: 4초
* 📦 Job ID: [2], [3]
* 🔍 해석:
  + 초기 df.show() 실행으로 실제 Spark Job이 수행됨
  + 이후 Statement 6에서 같은 데이터 재사용 → 빠르게 처리됨 (Query ID 1)

**📈 성능 및 실행 전략 진단**

| **항목** | **상태** | **해석** |
| --- | --- | --- |
| 총 SQL 실행 수 | 5개 | 간단한 탐색 또는 EDA 수준 |
| 평균 실행 시간 | 약 1~2초 | Spark 환경에서 매우 빠름 |
| 중복 쿼리 캐싱 | 있음 (Query ID 1) | 효율적 캐시 사용 확인됨 |
| 다단계 Job 발생 | 있음 (Query ID 4) | .toPandas()는 반드시 병렬 + 수집 + 변환으로 분리됨 |
| 리소스 병목 | 없음 | 앞서 분석한 Executor 상태와 일치 |

**✅ 결론**

이번 SQL / DataFrame 작업 흐름은 Spark의 **lazy evaluation** 및 **캐시 최적화**가 잘 작동하고 있으며, 쿼리 수는 적고 실행 시간도 빠른 상태입니다. 대부분은 데이터 확인, 변환, 수집용 명령입니다.

Description중 하나를 클릭하면 **Details for Query** 내용을 볼 수 있다

Query ID = 0인 쿼리의 **실행 세부 정보**, **실행 계획**, **관련 Job**, **쿼리 성능 메트릭**, **쿼리 결과 요약** 등을 제공합니다.

텍스트, 스크린샷, 폰트, 문서이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

**일반적으로 포함된 항목:**

| **섹션** | **설명** |
| --- | --- |
| **SQL Statement** | 실행된 SQL 쿼리 원문 |
| **Execution Plan (계획)** | Logical, Physical, Optimized Plan 확인 가능 |
| **Jobs** | 이 쿼리와 연관된 Spark Job 목록 (예: [2][3]) |
| **Stages/Tasks** | 각 Job의 Stage/Task 분할 구조 |
| **Metrics Summary** | Input size, Shuffle, Spill, Duration 등 |
| **Query Result Preview** | 쿼리 결과 일부 미리 보기 |
| **Error/Failure Info** | 실패 시 에러 메시지와 스택 트레이스 |

**🔍 현재 Query ID 0의 맥락 정리**

**🔸 제출 시간:**

* 2025/04/04 22:19:33

**🔸 Duration:**

* 4초 → Spark 기준으로는 비교적 가벼운 쿼리

**🔸 연관된 Job:**

* [2][3]
  + Job 2: 데이터 읽기 (executor 실행)
  + Job 3: 출력 처리 (예: .show())

**🔸 추정 SQL 동작:**

* **Job group for statement 4**
* Spark Stage 로그에서 보면 이 시점에 .show()와 관련된 작업이 있으므로, 해당 SQL 쿼리는 아마도:

sql

SELECT \* FROM some\_table LIMIT 10

또는

df.show()

와 같이 **DataFrame 또는 SQL 결과를 화면에 출력하기 위한 쿼리**로 추정됩니다.

**📈 예상되는 실행 계획 (Plan)**

실제 화면에서는 다음과 같은 구조로 보여집니다:

== Physical Plan ==

CollectLimit 10

+- \*(1) Project [...fields...]

+- \*(1) FileScan parquet ... [columns]

| **연산자** | **설명** |
| --- | --- |
| FileScan | 실제 데이터를 S3 등에서 읽는 작업 |
| Project | 필요한 컬럼만 추출 |
| CollectLimit | LIMIT 10과 같은 제한 적용 |
| Collect | 데이터를 Driver로 모음 (예: .show(), .toPandas() 등) |

**✅ 해석 요약**

| **항목** | **내용** |
| --- | --- |
| Query Type | 탐색 또는 출력용 (df.show() 또는 LIMIT 쿼리) |
| 성능 상태 | 빠르게 완료됨 (4초) |
| 관련 작업 | Job 2, Job 3에서 실행 완료 |
| 실행 계획 | FileScan → Project → Limit → Collect |
| 최적화 여부 | 스테이지 수 적고 shuffle 없음 → 최적화 양호 |

**🔧 필요 시 다음을 확인해보세요:**

| **확인 항목** | **이유** |
| --- | --- |
| **Physical Plan** | 실제 연산 순서, 병목 원인 확인 가능 |
| **Job → Stage → Task** | 작업 분산 상태, 느린 태스크 존재 여부 분석 |
| **Shuffle 여부** | Join, GroupBy, Sort 사용 여부 판단 |
| **쿼리 결과 데이터** | 데이터량 확인 → toPandas 등 다음 연산 계획 가능 |

Spark SQL에서 말하는 Physical Plan (물리 실행 계획)은 쿼리가 실제로 어떻게 실행되는지를 보여주는 **Spark 실행 엔진의 로우레벨 연산 플랜**입니다.

**🧩 Physical Plan이란?**

Spark SQL은 쿼리를 다음 3단계로 처리합니다:

1. **Logical Plan**
   * SQL 또는 DataFrame 코드의 의미론적 구조
   * 예: SELECT name FROM users WHERE age > 20
2. **Optimized Logical Plan**
   * Catalyst Optimizer가 최적화 적용 (필터 푸시다운, 컬럼 제거 등)
3. ✅ **Physical Plan**
   * 실제 Spark 작업 단위로 번역된 실행 로직
   * Spark가 Job/Stage/Task로 분할해 실행하는 단위
   * 여기서 **FileScan, Project, Filter, HashAggregate, SortMergeJoin** 같은 구체적인 연산자들이 등장함

**🔍 예시: .show() 또는 SELECT \* FROM table LIMIT 10**

== Physical Plan ==

CollectLimit 10

+- \*(1) Project [id, name, age]

+- \*(1) FileScan parquet [id, name, age] ...

| **연산자** | **의미** |
| --- | --- |
| FileScan | 데이터를 실제로 읽는 연산 (ex. S3 Parquet, CSV 등) |
| Project | 필요한 컬럼만 선택 (SELECT id, name) |
| CollectLimit | 드라이버로 10개만 모음 (LIMIT 10 or .show()) |
| \*(1) | 병렬 Task를 사용하는 Stage ID 표시 (여러 연산이 같은 Stage에서 실행됨) |

**🔍 예시: Join 쿼리**

== Physical Plan ==

SortMergeJoin [id], [user\_id]

:- \*(1) Sort [id ASC]

: +- \*(1) FileScan parquet [id, name]

+- \*(2) Sort [user\_id ASC]

+- \*(2) FileScan parquet [user\_id, score]

| **연산자** | **의미** |
| --- | --- |
| SortMergeJoin | 두 테이블을 Join할 때 정렬 기반 병합 방식 사용 |
| FileScan | 두 테이블의 데이터를 개별적으로 읽음 |
| Sort | Join 전에 키 기준 정렬 수행 |

**🔧 해석 팁**

| **패턴** | **의미 / 주의점** |
| --- | --- |
| FileScan | 데이터 읽기 단계 — Input 사이즈 확인 |
| Project | 컬럼 제한 — 쓸데없는 컬럼 읽으면 비효율 |
| CollectLimit, Collect | Driver 메모리로 가져옴 — **데이터 크기 주의 (OOM 위험)** |
| SortMergeJoin, BroadcastHashJoin | Join 종류 — 성능 차이 큼 |
| Exchange | Shuffle 발생 시점 — 네트워크 비용 있음 |
| Aggregate, HashAggregate | 집계 연산 — skew 여부 확인 필요 |

**✅ 결론: Physical Plan은 이런 걸 알려줍니다**

| **질문** | **Physical Plan으로 확인 가능?** |
| --- | --- |
| 데이터를 어디서 어떻게 읽는가? | ✅ FileScan |
| 연산이 어디서 일어나는가? | ✅ Stage 내 연산자 구성 |
| Join/GroupBy에서 shuffle이 생기는가? | ✅ Exchange 존재 여부 |
| 성능 병목 지점은 어디인가? | ✅ 연산자별 Task 시간 분석 |

Getting-started-emr-serverless.ipynb 파이썬 소스 설명

**📘 개요**

* **노트북 제목**: Get started with EMR Serverless on EMR Studio
* **주요 내용**:
  1. Spark 세션 구성
  2. 시각화를 위한 라이브러리 가져오기
  3. S3에서 데이터 읽기 및 처리
  4. Spark SQL 사용
  5. 시각화

**🧱 사전 준비**

* **필수 조건**:
  + Compute로 EMR Serverless 선택
  + Studio user role에 적절한 권한 필요 (Application 연결, role 전달)
  + PySpark 커널 사용
  + 인터넷 연결 가능한 VPC 설정 필요

**Spark 세션 구성**

**셀 1 :**

%%configure -f

{

"conf": {

"spark.pyspark.virtualenv.enabled": "true",

"spark.pyspark.virtualenv.bin.path": "/usr/bin/virtualenv",

"spark.pyspark.virtualenv.type": "native",

"spark.pyspark.python": "/usr/bin/python3",

"spark.executorEnv.PYSPARK\_PYTHON": "/usr/bin/python3"

}

}

* Spark에서 가상환경(Virtualenv) 설정
* PyPI 패키지를 설치 가능하게 하려는 목적

**셀 2: spark**

spark

* 현재 Spark 세션 객체가 정상인지 확인 (출력은 없음)

**셀 3: %%info**

%%info

* 현재 세션 정보 및 Spark UI 링크 표시

**PyPI 패키지 설치**

**셀 4:**

sc.install\_pypi\_package("matplotlib")

* matplotlib 설치 (데이터 시각화를 위한 패키지)

**S3에서 데이터 읽기**

**셀 5:**

file\_name = "s3://athena-examples-us-east-1/notebooks/yellow\_tripdata\_2016-01.parquet"

taxi\_df = (spark.read.format("parquet").option("header", "true")

.option("inferSchema", "true").load(file\_name))

* S3에서 NYC 택시 데이터셋을 읽음
* Parquet 포맷이고, 헤더 포함 및 스키마 자동 추론

**셀 6:**

taxi1\_df = taxi\_df.groupBy("VendorID", "passenger\_count").count()

taxi1\_df.show()

* VendorID와 승객 수별로 그룹핑해서 카운트

**셀 7:**

%%display

taxi1\_df

* 결과를 시각적으로 보기 위한 매직 명령
* 표 또는 차트로 확인 가능

**Spark SQL 사용**

**셀 8:**

taxi\_df.createOrReplaceTempView("taxis")

sqlDF = spark.sql(

"SELECT DOLocationID, sum(total\_amount) as sum\_total\_amount \

FROM taxis where DOLocationID < 25 Group by DOLocationID ORDER BY DOLocationID"

)

sqlDF.show(50)

* 임시 뷰 생성 (taxis)
* DOLocationID가 25 미만인 행에 대해 total\_amount 합산

**셀 9:**

%%sql

SHOW DATABASES

* 사용 가능한 데이터베이스 목록 출력

**Python으로 시각화**

**셀 10:**

import matplotlib.pyplot as plt

import numpy as np

import pandas as pd

plt.clf()

df = sqlDF.toPandas()

plt.bar(df.DOLocationID, df.sum\_total\_amount)

%matplot plt

* SQL 결과를 Pandas DataFrame으로 변환
* DOLocationID별 sum\_total\_amount를 바 차트로 시각화

AWS Cloud 및 빅데이터 처리 환경에서 자주 사용되는 데이터 파일 형식들은 성능, 압축 효율, 분산 처리 적합성 등의 이유로 선택된다. 각 형식의 특징과 사용 목적은 아래에 같다

**📁 1. Parquet**

* **형식**: **컬럼 기반** 저장 형식 (Columnar Storage)
* **특징**:
  + 특정 컬럼만 읽을 수 있어 I/O 효율이 좋음
  + **압축 효율이 높아 저장 공간 절약**
  + Apache Arrow 및 Spark, Athena, Redshift Spectrum과 호환
* **용도**:
  + 데이터 분석, ML 파이프라인, 대용량 데이터 저장
  + S3 + Athena 조합에서 쿼리 비용 절감

**📁 2. Avro**

* **형식**: **로우 기반** 저장 형식 (Row-Oriented)
* **특징**:
  + 스키마 내장 → 데이터와 함께 JSON 기반의 스키마 저장
  + **빠른 직렬화**/역직렬화
  + Kafka, Hadoop, Spark 등과 잘 호환됨
* **용도**:
  + Kafka를 통한 스트리밍 데이터 저장
  + 스키마가 자주 변할 수 있는 환경

**📁 3. ORC (Optimized Row Columnar)**

* **형식**: 컬럼 기반 저장 형식
* **특징**:
  + Parquet보다 더 높은 압축률 (특히 Hive + Tez에서 최적화됨)
  + 인덱싱과 스킵 기능 내장으로 빠른 쿼리 가능
* **용도**:
  + Hive, Presto, Amazon EMR 기반의 데이터 웨어하우징

**📁 4. JSON / JSONL**

* **형식**: 텍스트 기반, 계층적 구조 지원
* **특징**:
  + 사람이 읽기 쉬움
  + 구조가 유연하지만, 대규모 분석엔 비효율적
  + JSONL은 줄마다 하나의 JSON 객체가 있어 스트리밍에 적합
* **용도**:
  + 로그 파일, 설정 파일, Lambda 함수 간 데이터 전달
  + S3 + Glue ETL의 Raw 데이터로 자주 사용됨

**📁 5. CSV (Comma-Separated Values)**

* **형식**: 텍스트 기반, 로우 기반
* **특징**:
  + 매우 간단하고 범용적
  + 스키마 없음 (Glue나 Spark에서 스키마 추론 필요)
  + 데이터 크기가 크면 성능 저하
* **용도**:
  + 초기 데이터 로딩
  + 간단한 데이터셋 공유나 저장

**📁 6. Delta Lake**

* **형식**: Parquet + 트랜잭션 로그
* **특징**:
  + ACID 트랜잭션 지원
  + 스키마 진화 및 타임트래블 쿼리 가능
  + Apache Spark 기반
* **용도**:
  + 실시간 데이터 파이프라인 (Databricks 등에서 많이 사용)
  + EMR에서 Hudi/Iceberg와 함께 대체 사용 가능

**📁 7. Iceberg / Hudi**

* **형식**: 메타데이터 관리형 테이블 형식 (Apache 프로젝트)
* **특징**:
  + 테이블 수준 트랜잭션 및 증분 처리 지원
  + 대규모 테이블에서 효과적 (데이터레이크 테이블 포맷)
* **용도**:
  + Athena, EMR, Redshift Spectrum 등에서 실시간/증분 데이터 분석

**✅ 요약 비교표**

| **형식** | **저장 구조** | **장점** | **단점** | **주요 사용처** |
| --- | --- | --- | --- | --- |
| **Parquet** | Columnar | 빠른 분석, 압축 | 쓰기 성능↓ | Athena, EMR |
| **Avro** | Row | 스키마 내장, 빠른 직렬화 | 사람이 읽기 어려움 | Kafka, Glue |
| **ORC** | Columnar | Hive 최적화, 높은 압축 | Hive 중심 | Hive, EMR |
| **JSON** | Text | 유연성, 읽기 쉬움 | 비효율적 | Logs, Lambda |
| **CSV** | Text | 단순함 | 스키마 없음 | 초반 로딩 |
| **Delta** | Columnar + log | ACID 지원 | Databricks 특화 | ML 파이프라인 |
| **Iceberg/Hudi** | Table 포맷 | 증분 처리, 실시간 분석 | 설정 복잡 | S3 + Athena, EMR |

ACID 트랜잭션은 **데이터베이스나 데이터 레이크**에서 **신뢰할 수 있는 데이터 처리**를 보장하기 위해 사용하는 개념입니다.  
특히, **Delta Lake, Apache Hudi, Iceberg** 같은 최신 데이터 레이크 포맷에서 매우 중요하게 사용됩니다.

**✅ ACID 트랜잭션이란?**

ACID는 다음 4가지 속성의 약자입니다:

| **속성** | **설명** | **예시** |
| --- | --- | --- |
| **A - Atomicity (원자성)** | 트랜잭션은 **모두 실행되거나, 전혀 실행되지 않아야 함** | 데이터 입력 중 오류가 나면 전체 취소됨 |
| **C - Consistency (일관성)** | 트랜잭션 실행 전후에 **데이터가 항상 유효한 상태**여야 함 | 제약조건(예: unique) 위배 안됨 |
| **I - Isolation (격리성)** | 여러 트랜잭션이 동시에 수행돼도 **서로 간섭하지 않음** | 두 사용자가 같은 데이터를 동시에 수정해도 충돌 없음 |
| **D - Durability (지속성)** | 트랜잭션이 완료되면 **그 결과는 영구적으로 저장됨** | 시스템 꺼져도 데이터 보존 |

**💡 왜 ACID가 중요할까?**

**예시 1: 은행 송금**

* A 계좌 → B 계좌로 100만원 이체
* 중간에 시스템 오류가 나도 **돈이 증발하거나 중복 송금**되면 안 됨
  + ✅ 원자성: 둘 다 성공 or 둘 다 실패
  + ✅ 일관성: 잔액 합계는 변하지 않음
  + ✅ 격리성: 동시에 여러 송금이 있어도 꼬이지 않음
  + ✅ 지속성: 이체가 끝나면 시스템 꺼져도 기록 남음

**📦 데이터레이크에서의 ACID는?**

S3 같은 오브젝트 스토리지에는 원래 ACID 보장이 없지만,  
**Delta Lake / Hudi / Iceberg** 같은 테이블 포맷이 ACID 트랜잭션 기능을 제공해줍니다.

| **기술** | **ACID 지원 방식** | **주요 기능** |
| --- | --- | --- |
| **Delta Lake** | 트랜잭션 로그 \_delta\_log 사용 | 타임트래블, 스키마 진화 |
| **Apache Hudi** | commit log + version 관리 | 증분 처리, UPSERT |
| **Apache Iceberg** | snapshot 기반 메타데이터 관리 | 파티션 없는 테이블도 효율적 관리 |

**📌 결론**

* **ACID 트랜잭션**은 데이터 정합성과 신뢰성을 보장합니다.
* AWS에서는 Glue + S3, EMR, Athena, Redshift Spectrum 등에서 **Iceberg** 또는 **Hudi**와 함께 사용하면 데이터레이크에서도 ACID를 활용할 수 있습니다.

**AWS Glue MetaStore 통합**

emrs-interactive-app-admin-user 계정이 아닌 원래 계정으로 실습한다

CLI를 사용하여 작업을 제출시 cloud9에서 환경변수들을 설정하고 진행한다

|  |
| --- |
| export JOB\_ROLE\_ARN=arn:aws:iam::891377038690:role/EMRServerlessS3RuntimeRole  export S3\_BUCKET=s3://emrserverless-workshop-891377038690  export APPLICATION\_ID=00frg7agk4okfk09  export ACCOUNT\_ID=891377038690 |

Amazon S3->버킷->emrserverless-workshop-891377038690->taxi-data-glue

폴더가 이미 생성되어 있으므로 이를 삭제하고 진행한다(그냥 진행하면 : Failed)

CLI를 사용하여 작업을 제출을 수행하기 전에

EMR Studio->Applications->my-serverless-interactive-application->Spark-ETL-Glue-Metastore 에서 Spark properties 값은 아래와 같이 나온다

텍스트, 폰트, 라인, 스크린샷이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

CLI를 사용하여 작업을 제출을 수행한 후의 Spark properties

텍스트, 스크린샷, 폰트이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

파이썬 소스 분석

[s3://aws-data-analytics-workshops/emr-eks-workshop/scripts/spark-etl-glue.py](https://s3.console.aws.amazon.com/s3/buckets/aws-data-analytics-workshops/emr-eks-workshop/scripts/spark-etl-glue.py)

|  |
| --- |
| import sys  from datetime import datetime  from pyspark.sql import SparkSession  from pyspark.sql import SQLContext  from pyspark.sql.functions import \*  if \_\_name\_\_ == "\_\_main\_\_":  print(len(sys.argv))  if (len(sys.argv) != 4):  print("Usage: spark-etl-glue [input-folder] [output-folder] [dbName]")  sys.exit(0)  spark = SparkSession\  .builder\  .appName("Python Spark SQL Glue integration example")\  .enableHiveSupport()\  .getOrCreate()  nyTaxi = spark.read.option("inferSchema", "true").option("header", "true").csv(sys.argv[1])  updatedNYTaxi = nyTaxi.withColumn("current\_date", lit(datetime.now()))  updatedNYTaxi.printSchema()  print(updatedNYTaxi.show())  print("Total number of records: " + str(updatedNYTaxi.count()))    updatedNYTaxi.write.parquet(sys.argv[2])  updatedNYTaxi.registerTempTable("ny\_taxi\_table")  dbName = sys.argv[3]  spark.sql("CREATE database if not exists " + dbName)  spark.sql("USE " + dbName)  spark.sql("CREATE table if not exists ny\_taxi\_parquet USING PARQUET LOCATION '" + sys.argv[2] + "' AS SELECT \* from ny\_taxi\_table ")  spark.stop() |

이 PySpark 스크립트는 **CSV 데이터를 읽어와 처리한 뒤, Parquet 포맷으로 저장하고 Hive 테이블로 등록**하는 **ETL(Extract, Transform, Load)** 작업을 수행합니다.

**🔧 1. 의존성 및 초기 설정**

import sys

from datetime import datetime

from pyspark.sql import SparkSession

from pyspark.sql import SQLContext

from pyspark.sql.functions import \*

* **sys**: 명령줄 인자를 받기 위해 사용.
* **datetime**: 현재 시간을 컬럼에 추가하기 위해 사용.
* **pyspark.sql 관련 모듈들**: SparkSession을 생성하고 SQL 및 DataFrame 관련 기능을 제공.

**🏁 2. main 함수 시작 & 인자 확인**

print(len(sys.argv))

if (len(sys.argv) != 4):

print("Usage: spark-etl-glue [input-folder] [output-folder] [dbName]")

sys.exit(0)

* 명령줄 인자를 확인합니다.
* 입력 인자는 총 **3개** 받아야 합니다:
  1. input-folder: CSV 데이터가 있는 경로
  2. output-folder: Parquet 파일을 저장할 경로
  3. dbName: Hive에 생성할 데이터베이스 이름

**⚙️ 3. SparkSession 생성**

spark = SparkSession\

.builder\

.appName("Python Spark SQL Glue integration example")\

.enableHiveSupport()\

.getOrCreate()

* Hive 연동이 가능한 **SparkSession**을 생성합니다.

**📥 4. CSV 데이터 읽기**

nyTaxi = spark.read.option("inferSchema", "true").option("header", "true").csv(sys.argv[1])

* 입력받은 CSV 파일을 **스키마 자동 추론 및 헤더 사용** 옵션으로 읽어옵니다.

**🧪 5. 컬럼 추가 (현재 날짜)**

updatedNYTaxi = nyTaxi.withColumn("current\_date", lit(datetime.now()))

* current\_date라는 컬럼에 현재 시간 값을 추가합니다.

**🧾 6. 출력 및 레코드 수 확인**

updatedNYTaxi.printSchema()

print(updatedNYTaxi.show())

print("Total number of records: " + str(updatedNYTaxi.count()))

* 스키마 출력
* 데이터 출력
* 총 레코드 수 출력

**💾 7. Parquet 저장**

updatedNYTaxi.write.parquet(sys.argv[2])

* 데이터를 **Parquet 포맷**으로 지정된 경로에 저장합니다.

**🧮 8. 임시 테이블 등록**

updatedNYTaxi.registerTempTable("ny\_taxi\_table")

* Spark SQL에서 사용할 수 있도록 **임시 테이블 등록**

**🗃️ 9. Hive 데이터베이스 및 테이블 생성**

dbName = sys.argv[3]

spark.sql("CREATE database if not exists " + dbName)

spark.sql("USE " + dbName)

spark.sql("CREATE table if not exists ny\_taxi\_parquet USING PARQUET LOCATION '" + sys.argv[2] + "' AS SELECT \* from ny\_taxi\_table ")

* 지정된 dbName이 없으면 생성
* 해당 데이터베이스로 이동
* **ny\_taxi\_parquet 테이블 생성**  
  → 저장된 Parquet 파일을 Hive 테이블로 연결

**🛑 10. Spark 종료**

spark.stop()

**📌 요약**

| **단계** | **설명** |
| --- | --- |
| 입력 | CSV 파일에서 데이터 읽기 |
| 변환 | current\_date 컬럼 추가 |
| 출력 | Parquet 파일로 저장 |
| Hive 등록 | Hive에 데이터베이스 및 테이블 생성 및 연결 |

AWS Glue에서 사용하는 **AWS Data Catalog**는 실제로 **S3에 저장되지 않고**, **AWS 내부의 관리형 메타데이터 서비스**로 관리됩니다.  
즉, **Glue 테이블/데이터베이스 정보는 S3가 아닌 AWS의 내부 시스템에 저장**됩니다.

**🔍 AWS Data Catalog의 저장 위치 요약**

| **항목** | **저장 위치** | **설명** |
| --- | --- | --- |
| **테이블/데이터베이스 메타데이터** | **AWS Glue Data Catalog (AWS 내부 서비스)** | RDS 같은 AWS 관리형 서비스에 저장되며, 사용자는 접근 불가 (콘솔/SDK/API로만 접근 가능) |
| **실제 데이터 (예: Parquet 파일)** | **Amazon S3** | Glue 테이블이 참조하는 실제 데이터 파일 위치. 예: s3://my-bucket/output-folder/ |
| **크롤러 로그, 작업 로그** | Amazon S3 또는 CloudWatch |  |

AWS Glue Data Catalog는 사실상 AWS Glue의 **메타스토어(Metastore)** 역할을 하는 서비스이다. (실제 데이터는 S3에 저장하고 Metastore는 내부 Data Caralog에 저장 )

**🔍 AWS Glue Metastore란?**

**✅ 정의:**

**AWS Glue Metastore**는 **Apache Hive-compatible Metastore**로,  
**테이블, 데이터베이스, 스키마, 파티션 등의 메타데이터를 저장**하고 관리하는 역할을 합니다.

즉, AWS Glue의 **Data Catalog = Hive Metastore 역할**

**🧠 구성요소 이해**

| **구성 요소** | **설명** |
| --- | --- |
| **Database** | 테이블들의 논리적 그룹. Hive에서의 데이터베이스 개념과 동일 |
| **Table** | 실제 데이터 파일(S3 등)을 설명하는 메타데이터 (스키마, 위치 등) |
| **Partition** | 테이블 내부에서 데이터를 분할하여 저장할 때 사용 (예: 날짜별) |
| **Schema** | 각 컬럼의 데이터 타입 및 이름 등의 정의 |
| **Location** | 데이터가 실제로 저장된 S3 경로 |

**🔁 Hive Metastore와의 관계**

| **항목** | **Hive Metastore** | **AWS Glue Data Catalog** |
| --- | --- | --- |
| 저장 위치 | 자체 RDB (MySQL, PostgreSQL 등) | AWS 관리형 (내부 메타스토어) |
| 접근 방식 | 직접 JDBC 접근 가능 | AWS Console, Boto3, Glue SDK 등 |
| 관리 주체 | 사용자 | AWS가 완전 관리형으로 제공 |
| 통합 가능 | Spark, Hive, Presto 등 | Spark, Athena, Redshift, EMR 등과 연동 가능 |

**✅ 사용 예시: Spark에서 Glue Metastore 사용하기**

Spark에서 AWS Glue를 Metastore로 사용하려면 설정이 필요합니다:

spark = SparkSession.builder \

.appName("Glue Metastore Example") \

.config("hive.metastore.client.factory.class",

"com.amazonaws.glue.catalog.metastore.AWSGlueDataCatalogHiveClientFactory") \

.enableHiveSupport() \

.getOrCreate()

이렇게 하면 Spark SQL이 Glue Data Catalog를 **Hive Metastore처럼 사용**할 수 있다.

**☁️ Glue Metastore 활용 사례**

1. **Athena** → Glue Data Catalog에 등록된 테이블을 쿼리
2. **Spark on EMR/EC2** → Glue Metastore를 통해 메타데이터 공유
3. **Redshift Spectrum** → Glue의 테이블 메타정보로 외부 테이블 연결
4. **AWS Glue Jobs** → 내부적으로 Glue Metastore를 사용하여 테이블 참조

**🛡️ 요약**

| **질문** | **답변** |
| --- | --- |
| Glue Metastore가 어디에 있는가? | AWS 내부의 **Glue Data Catalog** |
| Hive Metastore와 같나? | **기능적으로 동일**하며, Spark에서 Hive Metastore처럼 사용 가능 |
| 데이터는 어디에? | 메타데이터는 Glue에, 실제 데이터는 보통 **S3**에 있음 |

**RDS Hive MetaStore 통합**

emrs-interactive-app-admin-user 계정이 아닌 원래 계정으로 실습한다

**✅ NAT Gateway 삭제**

이 실습 CloudFormation 템플릿은 **2개의 NAT Gateway**를 만들고 있고, 각각 Elastic IP(EIP)를 필요로 한니다.

- AWS는 기본적으로 계정당 **EIP 5개 제한**이 걸려 있다.

- 이미 EIP가 5개 이상 할당되어 있거나, 이전에 생성된 NAT Gateway가 남아 있다면, 이 제한을 초과하게 된다.

이미 앞 실습에서 5개의 EIP를 사용 중이므로 2개의 NAT Gateway를 삭제해야 한다

EC2->탄력적 IP에서 확인 가능

**텍스트, 스크린샷, 폰트, 번호이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.**

바로 앞의 AWS Glue MetaStore 통합 실습에서 사용했던 “emr-serverless-interactive” 스택을 삭제한다

혹시 DELETE\_FAILED가 나오면 앞 실습에서 생성된 아래 S3버킷을 수동으로 삭제(버킷 비우기 수행 후 삭제)하고 다시 스택 삭제 재시도(전체 스택 강제 삭제)를 누른다

Amazon S3 -> 버킷 -> **emrserverless-interactive-blog-891377038690-us-east-1**

MariaDB 버전 문제

실습의 template 사용시 오류 발생 : RDS MariaDB 버전 10.6.8이 현재 사용 불가하거나 지원되지 않음 (template 스크립트 수정 필요)

|  |
| --- |
| Resource handler returned message: "Cannot find version 10.6.8 for mariadb (Service: Rds, Status Code: 400, Request ID: e30bb027-07c6-4fc1-a295-3f4bd79be4b8) (SDK Attempt Count: 1)" (RequestToken: c37081e1-dbdc-67c4-3811-e3213d23e20d, HandlerErrorCode: InvalidRequest) |

Clodu9에서 아래 명령으로 MariaDB 지원되는 버전 확인 가능

aws rds describe-db-engine-versions \

--engine mariadb \

--query "DBEngineVersions[].EngineVersion" \

--region us-east-1

아래 파일 다운로드 하여 수정해서 사용한다

https://aws-data-analytics-workshops.s3.amazonaws.com/emr-serverless-workshop/cloudformation/hive\_update\_metastore\_cft.yaml

다운로드된 파일에서 EngineVersion '10.6.8'을 찾아 지원 버전인 **‘10.6.14’**로 변경한다

|  |
| --- |
| VPCSecurityGroups:  - Ref : RDSIngressSecurityGroup  DBSubnetGroupName: !Ref RDSDBSubnetGroup  PubliclyAccessible: 'false'  **EngineVersion: '10.6.8' 🡨 10.6.14**  Tags: |

DBInstanceClass: 'db.t2.small' 을 찾아 'db.t3.small'로 변경한 다음 저장한다

|  |
| --- |
| Properties:  DBInstanceIdentifier: EMRServInstance  AllocatedStorage: '5'  **DBInstanceClass: 'db.t2.small'**  Engine: mariadb |

수정된 yaml 파일을 사용하여 CloudFromation에서 실습에서 지시된 내용과 다르게

[템플릿 파일 업로드]를 선택하여 직접 파일 업로드하여 진행한다

텍스트, 스크린샷, 폰트, 번호이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

계속 진행하기 전에 다음을 먼저 수행한다

**✅ CloudFormation 에서 stack 생성 전에 EC2 키 페어 생성 후 실습을 진행 한다**

1. **AWS Console -> EC2 -> 키 페어(Key Pairs)** 로 이동
2. **[키 페어 생성(Create key pair)]** 클릭
3. 이름 : **emr-hms-key**
4. **프라이빗 키 파일 형식:**  .pem으로 선택

텍스트, 스크린샷, 번호, 폰트이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

1. 키페어 생성 버튼 클릭하면 키파일이 자동 다운로드 됨
2. CloudFormation 스택 생성 시 KeyName으로 해당 키 이름 선택
3. 다운로드한 .pem 파일은 **잃어버리면 복구 불가**, 안전한 곳에 보관

**❗ 왜 꼭 필요할까?**

실습의 템플릿은 아래 리소스들을 생성하며, 일부는 EC2 인스턴스를 포함한다:

* AWS::Cloud9::EnvironmentEC2 (Cloud9 IDE)
* AWS::EMR::Cluster (EMR 클러스터 - EC2 기반)

이런 자원은 키 페어 없이는 SSH 접속이 불가능하기 때문에, 키 페어가 필수로 요구된다.

텍스트, 스크린샷, 폰트, 번호이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

15~20분 정도 소요

cloud9 에서 실행

|  |
| --- |
| sudo yum install jq -y  export MariaDBHost=emrservinstance.czeysqe6yl7y.us-east-1.rds.amazonaws.com  export JOB\_ROLE\_ARN=arn:aws:iam::891377038690:role/EMR-Serverless-HMS-EMRServerlessJobRole-TUbAIu6NT4Ay  export S3\_BUCKET=emr-hive-us-east-1-891377038690  export SPARK\_APPLICATION\_ID=00frgpmfk125uq09  export SECRET\_ID=rds-users-credentials  export DBUSER=$(aws secretsmanager get-secret-value --secret-id $SECRET\_ID | jq --raw-output '.SecretString' | jq -r .MasterUsername)  export DBPASSWORD=$(aws secretsmanager get-secret-value --secret-id $SECRET\_ID | jq --raw-output '.SecretString' | jq -r .MasterUserPassword)  export JDBCDriverClass=org.mariadb.jdbc.Driver  export JDBCDriver=mariadb-connector-java.jar |

**spark-nyctaxi.py 소스 분석**

|  |
| --- |
| import sys  from datetime import datetime  from pyspark.sql import SparkSession  from pyspark.sql.functions import \*  if \_\_name\_\_ == "\_\_main\_\_":  print(len(sys.argv))  if (len(sys.argv) != 4):  print("Usage: spark-nyctaxi [warehouse-location] [input-folder] [output-folder]")  sys.exit(0)    print("Warehouse location: " + sys.argv[1]+"/warehouse/")  print("CSV folder path: " + sys.argv[2])  print("Writing the parquet file to the folder: " + sys.argv[3])    spark = SparkSession \  .builder \  .config("spark.sql.warehouse.dir", sys.argv[1]+"/warehouse/" ) \  .enableHiveSupport() \  .getOrCreate()    nyTaxi = spark.read.option("inferSchema", "true").option("header", "true").csv(sys.argv[2])    updatedNYTaxi = nyTaxi.withColumn("current\_date", lit(datetime.now()))    updatedNYTaxi.registerTempTable("ny\_taxi\_table")    spark.sql("SHOW DATABASES").show()  spark.sql("CREATE DATABASE IF NOT EXISTS `hivemetastore`")  spark.sql("DROP TABLE IF EXISTS hivemetastore.ny\_taxi\_parquet")    updatedNYTaxi.write.option("path",sys.argv[3]).mode("overwrite").format("parquet").saveAsTable("hivemetastore.ny\_taxi\_parquet") |

NYC Taxi CSV 데이터를 읽어서 Parquet 포맷으로 저장하고, Hive 테이블로 등록하는 ETL 처리 코드이다.  
Hive Metastore와 Spark SQL 통합을 사용하는 구조이므로 EMR / EMR Serverless + Hive Metastore 연동에 적합한 예제

**📘 전체 개요**

| **목적** | **설명** |
| --- | --- |
| 입력 | NYC Taxi CSV 파일 |
| 처리 | 현재 날짜 컬럼 추가 |
| 출력 | Parquet 형식으로 저장 |
| 메타데이터 | Hive Metastore에 테이블로 등록 (hivemetastore.ny\_taxi\_parquet) |

**📦 코드 분석**

**✅ 1. 인자 처리**

if (len(sys.argv) != 4):

print("Usage: spark-nyctaxi [warehouse-location] [input-folder] [output-folder]")

sys.exit(0)

* 실행 시 3개의 인자를 받아야 함:
  1. warehouse-location: Hive warehouse 디렉토리 (보통 S3 또는 HDFS)
  2. input-folder: NYC CSV 파일이 있는 경로
  3. output-folder: Parquet 파일 저장할 경로

spark-submit spark\_nyctaxi.py s3://my-bucket spark\_input/ spark\_output/

**✅ 2. SparkSession 생성**

spark = SparkSession \

.builder \

.config("spark.sql.warehouse.dir", sys.argv[1]+"/warehouse/") \

.enableHiveSupport() \

.getOrCreate()

* Hive Metastore 연동을 활성화하고, Warehouse 디렉토리 설정

**✅ 3. CSV 파일 읽기**

nyTaxi = spark.read.option("inferSchema", "true").option("header", "true").csv(sys.argv[2])

* NYC 택시 CSV 파일을 **스키마 자동 추론 + 헤더 사용** 옵션으로 읽음

**✅ 4. 현재 날짜 컬럼 추가**

updatedNYTaxi = nyTaxi.withColumn("current\_date", lit(datetime.now()))

* 모든 레코드에 현재 날짜를 나타내는 current\_date 컬럼 추가

**✅ 5. Temp View 등록 (Spark SQL용)**

updatedNYTaxi.registerTempTable("ny\_taxi\_table")

* Spark SQL로 임시 쿼리를 날릴 수 있도록 테이블 등록

**✅ 6. Hive Database/테이블 관리**

spark.sql("SHOW DATABASES").show()

spark.sql("CREATE DATABASE IF NOT EXISTS `hivemetastore`")

spark.sql("DROP TABLE IF EXISTS hivemetastore.ny\_taxi\_parquet")

* hivemetastore라는 Hive DB가 없으면 생성
* 이전에 같은 이름의 테이블이 있으면 삭제

**✅ 7. Parquet 파일로 저장 + Hive 테이블로 등록**

updatedNYTaxi.write \

.option("path", sys.argv[3]) \

.mode("overwrite") \

.format("parquet") \

.saveAsTable("hivemetastore.ny\_taxi\_parquet")

* 지정한 경로 (output-folder)에 **Parquet 형식**으로 저장
* 동시에 **Hive 테이블로 등록**
  + 테이블 이름: hivemetastore.ny\_taxi\_parquet
  + 경로는 지정된 --output-folder

**✅ 실행 후 결과**

* S3 (또는 로컬 디렉토리)에 Parquet 파일 생성
* Hive 메타스토어에 테이블 등록
* Spark SQL에서 다음과 같이 쿼리 가능:

SELECT \* FROM hivemetastore.ny\_taxi\_parquet LIMIT 10;

**🔧 응용 포인트**

| **활용 예** | **방법** |
| --- | --- |
| EMR Serverless 연동 | Hive Metastore가 RDS로 구성되어 있다면 정상 연동 가능 |
| Athena에서 쿼리 | 같은 S3 위치를 Glue Data Catalog에 등록하면 가능 |
| MWAA DAG에서 실행 | SparkSubmitOperator로 이 스크립트를 트리거 가능 |

**🔍 보완 아이디어**

* datetime.now() 대신 current\_timestamp() 사용하면 UTC 기준으로 Spark가 처리
* registerTempTable → createOrReplaceTempView()로 대체하는 것이 최신 API

MariaDB SQL 쿼리 추가 실습

SHOW TABLES;

SELECT \* from TBLS;

SELECT \* from DBS;

스크린샷, 텍스트, 라인, 소프트웨어이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

Hive Metastore의 내부 메타데이터 DB(RDS/MariaDB) 에서 등록된 테이블 목록을 데이터베이스별로 조회하는 쿼리

|  |
| --- |
| SELECT d.NAME AS database\_name, t.TBL\_NAME AS table\_name  FROM TBLS t  JOIN DBS d ON t.DB\_ID = d.DB\_ID; |

<출력 결과>

텍스트, 스크린샷, 소프트웨어이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

특정 테이블의 저장 위치(S3 경로) 조회 쿼리

|  |
| --- |
| SELECT d.NAME AS database\_name, t.TBL\_NAME AS table\_name, s.LOCATION  FROM TBLS t  JOIN DBS d ON t.DB\_ID = d.DB\_ID  JOIN SDS s ON t.SD\_ID = s.SD\_ID  WHERE t.TBL\_NAME = 'ny\_taxi\_parquet'; |

텍스트, 스크린샷, 폰트, 소프트웨어이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

특정 테이블(ny\_taxi\_parquet)의 컬럼 구조(스키마) 를 조회

|  |
| --- |
| SELECT c.COLUMN\_NAME, c.TYPE\_NAME, c.INTEGER\_IDX  FROM TBLS t  JOIN SDS s ON t.SD\_ID = s.SD\_ID  JOIN CDS cd ON s.CD\_ID = cd.CD\_ID  JOIN COLUMNS\_V2 c ON cd.CD\_ID = c.CD\_ID  WHERE t.TBL\_NAME = 'ny\_taxi\_parquet'  ORDER BY c.INTEGER\_IDX; |

<출력 결과>

텍스트, 스크린샷이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

**🔍 실습 예제 쿼리 목적**

**Hive 테이블 ny\_taxi\_parquet에 등록된 컬럼명, 타입, 순서**를 조회합니다.

**🧱 각 테이블 설명**

| **테이블** | **설명** |
| --- | --- |
| TBLS | Hive에 등록된 **테이블 정보**를 저장하는 테이블 |
| SDS | 테이블의 **저장 포맷 및 위치(StorageDescriptor)** 정보를 가진 테이블 |
| CDS | 테이블의 **컬럼 정보 묶음(Column Descriptor)** 을 참조 |
| COLUMNS\_V2 | 실제 컬럼들의 이름, 타입, 순서 정보를 저장하는 테이블 |

**🔗 조인 흐름 설명**

TBLS (테이블 메타데이터)

└──→ SDS (Storage Descriptor) ← t.SD\_ID = s.SD\_ID

└──→ CDS (Column Descriptor) ← s.CD\_ID = cd.CD\_ID

└──→ COLUMNS\_V2 (컬럼 정보) ← cd.CD\_ID = c.CD\_ID

* Hive에서 테이블을 생성하면, 컬럼 스키마는 이렇게 다단계로 저장됩니다.
* 하나의 테이블 → 하나의 StorageDescriptor → 하나의 ColumnDescriptor → 여러 COLUMNS\_V2 레코드

**✅ 결과 컬럼 설명**

| **컬럼명** | **의미** |
| --- | --- |
| COLUMN\_NAME | 컬럼 이름 (예: vendor\_id, pickup\_time) |
| TYPE\_NAME | 데이터 타입 (예: string, int, timestamp) |
| INTEGER\_IDX | 컬럼 순서 (0부터 시작) |

**🔍 결과 예시**

| **COLUMN\_NAME** | **TYPE\_NAME** | **INTEGER\_IDX** |
| --- | --- | --- |
| vendor\_id | int | 0 |
| pickup\_datetime | timestamp | 1 |
| dropoff\_datetime | timestamp | 2 |
| fare\_amount | double | 3 |

**🛠 실무 활용**

| **사용 목적** | **예** |
| --- | --- |
| 스키마 검증 | Spark 작업이나 Athena 테이블과 컬럼 일치 여부 확인 |
| 테이블 구조 문서화 | 자동 스키마 리포트 작성 |
| 컬럼 통계 수집 | 컬럼 수, 타입 종류 등 분석용 스크립트에 활용 |

**MWAA를 사용한 오케스트레이션**

원래 계정으로 실습

**앞 실습에서 생성한 스택 “EMR-Serverless-HMS양식의 맨 위”을 찾아 삭제한다(가용 EIP 개수를 확보)**

**S3 버킷이(emr-hive-us-east-1-891377038690)이 삭제되지 않을 경우 수동으로 삭제한다**

**실습에서 제공되는 template 사용시 AWS Managed Workflows for Apache Airflow(MWAA)에서 더 이상 Airflow 2.2.2 버전을 지원하지 않기 때문에 발생한 오류 발생**

**텍스트, 스크린샷, 폰트이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.**

**🔧 지원되는 최신 Airflow 버전 사용**

**AWS MWAA에서 현재(2025년 기준) 지원하는 버전은 다음 중 하나입니다:**

| **지원되는 버전 (예시)** |
| --- |
| **2.4.3 ✅ 추천** |
| **2.5.1 ✅ 안정** |
| **2.6.3 ✅ 최신 안정** |
| **2.7.2 ✅ 최신 기능 포함** |

Template 파일(airflow\_cft.yml)을 다운받아서 AirflowVersion: 2.2.2 🡪 AirflowVersion: 2.4.3으로 수정하고 저장한 다음 [템플릿 파일 업로드]로 선택하여 수정된 파일 업로드해서 스택을 생성한다

|  |
| --- |
| SecurityGroupIds:  - !Ref NoIngressSecurityGroup  WebserverAccessMode: PUBLIC\_ONLY  **AirflowVersion: 2.4.3**  LoggingConfiguration:  DagProcessingLogs: |

**텍스트, 스크린샷, 폰트, 번호이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.**

**스택 생성 완료까지 약 20~30분 소요되므로 기다린다**

**Amazon MWAA (Managed Workflows for Apache Airflow)**

**: Apache Airflow를 AWS에서 완전관리형(Managed)으로 제공하는 서비스입니다.  
복잡한 데이터 파이프라인을 직관적으로 정의하고 자동 실행할 수 있도록 AWS가 인프라 운영을 대신해주는 서비스.**

**🎯 Amazon MWAA란?**

| **항목** | **설명** |
| --- | --- |
| **정식 명칭** | **Amazon Managed Workflows for Apache Airflow (MWAA)** |
| **목적** | **Apache Airflow를 설치 없이 즉시 사용할 수 있도록 지원** |
| **대상** | **데이터 파이프라인, ETL, ML 워크플로 등을 운영하고자 하는 사용자** |
| **배포 방식** | **AWS가 자동으로 Airflow 실행 환경(웹서버, 스케줄러, 워커)을 구축 및 운영** |
| **주요 장점** | **서버 관리 없음, 보안 구성 쉬움, AWS 서비스와 원활한 통합** |

**🧱 구성 요소**

**Amazon MWAA는 다음의 Airflow 구성 요소들을 포함합니다:**

| **구성요소** | **역할** |
| --- | --- |
| **Web Server** | **DAG 및 실행 상태를 확인하는 UI 제공** |
| **Scheduler** | **DAG 스케줄링 및 트리거** |
| **Worker** | **각 Task 실행** |
| **Metadata DB** | **DAG 및 실행 이력 저장 (AWS에서 자동 관리)** |
| **Logging** | **CloudWatch Logs에 자동 저장** |
| **DAG 코드 저장소** | **S3 버킷에 DAG 코드 업로드하여 실행** |

**🗂️ 주요 설정 요소**

| **항목** | **설명** |
| --- | --- |
| **Airflow 버전** | **1.10.12, 2.0.2, 2.2.2, 2.4.3 등 선택 가능** |
| **DAG 코드 위치** | **S3 버킷 (s3://your-mwaa-bucket/dags/)** |
| **플러그인/라이브러리** | **S3 경로 지정 가능 (예: requirements.txt, plugins.zip)** |
| **환경 변수** | **DAG 내에서 사용할 ENV 설정 가능** |
| **IAM 권한** | **Task가 사용할 Execution Role 지정** |
| **VPC** | **반드시 VPC에 배치되어야 함 (Private Subnet 권장)** |

**🧰 주요 기능 및 특징**

**✅ Airflow 기본 기능 그대로**

* **DAG 작성, 의존성 설정, 태스크 실패 재시도, SLA 등 Airflow의 모든 기능 그대로 사용 가능**

**✅ AWS 서비스와 통합**

* **S3, RDS, Redshift, EMR, Glue, Lambda 등과 연동 쉬움**
* **예: S3ToRedshiftOperator, EmrAddStepsOperator 등 사용 가능**

**✅ 완전관리형 (Managed)**

* **EC2, ALB, DB 등 인프라 관리 불필요**
* **Auto-scaling 및 Auto-healing 지원**

**✅ 보안 연동 쉬움**

* **IAM Role (Execution Role) + KMS + VPC + Secrets Manager 등 통합**

**💡 DAG 실행 예시 (S3 → EMR → Slack 알림)**

1. **S3에 새로운 CSV 업로드 감지**
2. **Glue로 변환 또는 EMR에서 Spark 처리**
3. **처리 결과를 Redshift로 적재**
4. **Slack 또는 이메일로 결과 전송**

**→ 이 모든 흐름을 Airflow DAG 하나로 구성 가능**

**🔒 보안 및 네트워크**

* **VPC 필수 (보통 Private Subnet + NAT Gateway 사용)**
* **IAM Role로 각 태스크가 AWS 서비스 접근**
* **S3, CloudWatch, Secrets Manager, KMS 모두 통합 가능**

**🧪 요금 구조**

* **시간 단위로 실행된 환경 시간에 따라 요금 청구**
* **예: t3.medium 스케일에서 한 시간 작동하면 시간당 요금 발생**
* **CloudWatch 로그, S3 사용량, NAT 트래픽 등은 별도 청구**

**📦 예시 디렉토리 구조 (S3)**

s3://your-mwaa-bucket/

│

├── dags/

│ └── my\_etl\_dag.py

│

├── plugins/

│ └── my\_custom\_plugin.py

│

└── requirements.txt

**🛠️ 사용 절차 요약**

1. **S3 버킷 생성 → dags/, plugins/, requirements.txt 업로드**
2. **VPC, Subnet, Security Group 준비**
3. **IAM Execution Role 생성**
4. **MWAA 환경 생성 (Airflow 버전 선택)**
5. **DAG 업로드 → 자동 반영 → 실행/모니터링**

**✅ 요약**

| **항목** | **내용** |
| --- | --- |
| **핵심 기능** | **Apache Airflow 완전관리형 실행** |
| **주요 장점** | **인프라 관리 無, AWS 서비스 연동 최적화** |
| **DAG 저장** | **S3 버킷 내 dags/ 폴더** |
| **보안 구성** | **IAM + VPC + KMS + CloudWatch 완벽 통합** |
| **사용 대상** | **데이터 엔지니어, ML 파이프라인 운영자 등** |

Apache Airflow는 **워크플로(Workflow)** 를 정의하고, **스케줄링 및 모니터링** 할 수 있는 **오픈소스 플랫폼**입니다.  
복잡한 데이터 파이프라인을 **직관적으로 코드로 구성하고 실행**할 수 있도록 설계된 도구입니다.

**🎯 Apache Airflow란?**

| **항목** | **설명** |
| --- | --- |
| 📌 정의 | 워크플로 자동화 및 스케줄링을 위한 Python 기반 플랫폼 |
| 🏗️ 워크플로 표현 | Python 코드로 DAG(Directed Acyclic Graph)를 정의 |
| ⏱️ 실행 방식 | 각 Task를 다양한 시스템에서 병렬로 실행 가능 |
| 🔎 모니터링 | 웹 UI를 통해 실행 상태 시각적으로 확인 가능 |
| 🔌 확장성 | 다양한 시스템과 연동 (Spark, Hive, EMR, Kubernetes, S3 등) |

**🧱 기본 개념**

**1. DAG (Directed Acyclic Graph)**

워크플로의 전체 구조를 의미합니다.  
작업(Task) 간의 **의존성 관계를 가진 비순환 그래프**입니다.

**✅ DAG = Directed Acyclic Graph**

* **방향성(Directed)**: 실행 순서가 정해져 있음
* **비순환(Acyclic)**: 순환(Loop)이 없음 — 한 작업이 다시 자기 자신으로 돌아가지 않음
* 즉, **작업(Task) 간의 실행 순서와 의존 관계를 표현하는 그래프 구조**

**2. Task**

워크플로 내에서 실행되는 개별 작업입니다. 예:

* S3에서 파일 다운로드
* Spark job 실행
* Slack 알림 전송

**3. Operator**

각 Task의 실행 방법을 정의하는 템플릿입니다.  
예: PythonOperator, BashOperator, SparkSubmitOperator, EMRStepOperator, DummyOperator 등

**🔧 예시 코드 (DAG 정의 예)**

from airflow import DAG

from airflow.operators.bash import BashOperator

from datetime import datetime

with DAG(

dag\_id="sample\_workflow",

start\_date=datetime(2024, 1, 1),

schedule\_interval="@daily"

) as dag:

task1 = BashOperator(

task\_id="print\_date",

bash\_command="date"

)

task2 = BashOperator(

task\_id="say\_hello",

bash\_command="echo 'Hello Airflow'"

)

task1 >> task2 # task1이 끝난 후 task2 실행

**📊 웹 UI**

Airflow는 강력한 웹 UI를 제공합니다:

* DAG 목록 및 상태 확인
* DAG 실행 로그 보기
* Task 간 의존성 시각화
* 수동 실행 및 재시도 버튼 제공

**🛠️ 주요 사용처**

| **분야** | **설명** |
| --- | --- |
| ETL 파이프라인 관리 | 데이터 수집, 전처리, 저장을 단계별 자동화 |
| 머신러닝 파이프라인 | 모델 훈련, 평가, 배포 등 순차적 작업 관리 |
| DevOps 자동화 | 배치 작업, 시스템 점검, 자동 알림 |

[실습 계속]

스택 생성 완료 후 cloud9 인스턴스를 사용하지 말고

윈도우에서 매모장으로 requirements.txt 파일을 만든다

아래 내용 복사하고 저장한다

|  |
| --- |
| emr\_serverless @ https://github.com/aws-samples/emr-serverless-samples/releases/download/v0.0.4-preview/mwaa\_plugin.zip 양식의 맨 아래 |

아래 파이썬 소스도 윈도우에 다운받아서  **ApplicationID, JobRoleArn 및 S3Bucket** 의 값을 위의 CloudFormation 스택에서 적절한 섹션의 코드로 복사한 값으로 바꿉니다. https://aws-data-analytics-workshops.s3.amazonaws.com/emr-serverless-workshop/scripts/example\_emr\_serverless.py

|  |
| --- |
| import os  from datetime import datetime  from airflow import DAG  from emr\_serverless.operators.emr import EmrServerlessStartJobOperator  # Replace these values in square brackets with your correct values  APPLICATION\_ID = os.getenv("APPLICATION\_ID", "00frgtbdmdkjhr09")  JOB\_ROLE\_ARN = os.getenv("JOB\_ROLE\_ARN", "arn:aws:iam::891377038690:role/EMR-Serverless-Orchestration-EMRServerlessJobRole-I79FAZ8RKerH")  S3\_BUCKET = os.getenv("S3\_BUCKET", "airflow-us-east-1-891377038690")  # [START howto\_operator\_emr\_serverless\_config]  JOB\_DRIVER\_ARG = {  "sparkSubmit": {  "entryPoint": "local:///usr/lib/spark/examples/src/main/python/pi.py",  }  }  CONFIGURATION\_OVERRIDES\_ARG = {  "monitoringConfiguration": {  "s3MonitoringConfiguration": {  "logUri": f"s3://{S3\_BUCKET}/logs/"  }  },  }  # [END howto\_operator\_emr\_serverless\_config]  with DAG(  dag\_id='example\_emr\_serverless\_job',  schedule\_interval=None,  start\_date=datetime(2021, 1, 1),  tags=['example'],  catchup=False,  ) as dag:  job\_starter = EmrServerlessStartJobOperator(  task\_id="start\_job",  application\_id=APPLICATION\_ID,  execution\_role\_arn=JOB\_ROLE\_ARN,  job\_driver=JOB\_DRIVER\_ARG,  configuration\_overrides=CONFIGURATION\_OVERRIDES\_ARG,  )  # [END howto\_operator\_emr\_serverless\_job] |

[Apache Airflow UI에서 작업 트리거후 결과를 확인한다](https://catalog.us-east-1.prod.workshops.aws/workshops/e8e8fbb5-c3fb-4f86-bf77-0ba1fe402c55/en-US/orchestration/submit-spark-jobs" \l "trigger-the-jobs-in-apache-airflow-ui)

Amazon S3->버킷->airflow-us-east-1-891377038690->logs->applications->00frgtbdmdkjhr09->jobs->00frgu7dp4gt500b->SPARK\_DRIVER->stdout.gz로

이동해서 객체작업 -> “S3 Select를 사용한 쿼리”를 클릭하고 나머지 설정 그대로 두고서

[SQL 쿼리 실행]버튼을 누르면 쿼리 결과에 출력이 보여 진다 양식의 맨 위

텍스트, 폰트, 라인, 스크린샷이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

"entryPoint": "local:///usr/lib/spark/examples/src/main/python/pi.py"에 들어 있는

파이썬 코드 분석

|  |
| --- |
| from \_\_future\_\_ import print\_function  import sys  from pyspark.sql import SparkSession  from random import random  if \_\_name\_\_ == "\_\_main\_\_":  spark = SparkSession.builder.appName("PythonPi").getOrCreate()  partitions = int(sys.argv[1]) if len(sys.argv) > 1 else 2  n = 100000 \* partitions  def inside(\_):  x, y = random(), random()  return x\*x + y\*y < 1  count = spark.sparkContext.parallelize(range(1, n + 1), partitions) \  .filter(inside) \  .count()  print("Pi is roughly %f" % (4.0 \* count / n))  spark.stop() |

📌 **GitHub 공식 저장소 (Apache Spark)**  
링크:  
🔗 [**https://github.com/apache/spark/blob/master/examples/src/main/python/pi.py**](https://github.com/apache/spark/blob/master/examples/src/main/python/pi.py)

**🔍 코드 분석**

**이 코드는 Spark를 이용해 몬테카를로 시뮬레이션으로 π(파이) 값을 근사 계산하는 매우 유명한 예제입니다.**

**📌 1. SparkSession 생성**

spark = SparkSession.builder.appName("PythonPi").getOrCreate()

* Spark 애플리케이션 시작
* 앱 이름은 PythonPi로 설정

**📌 2. 파티션 수 및 반복 횟수 설정**

partitions = int(sys.argv[1]) if len(sys.argv) > 1 else 2

n = 100000 \* partitions

* 파티션 수를 명령줄 인자에서 받아옴 (없으면 기본 2)
* n은 시뮬레이션 반복 횟수 (기본 200,000회)

**📌 3. 원 안에 들어가는 점 판별 함수**

def inside(\_):

x, y = random(), random()

return x\*x + y\*y < 1

* 단위 정사각형(1×1) 내에서 무작위 점 (x, y)를 찍고
* 반지름 1의 원 안에 들어가는지를 판별
* x² + y² < 1이면 원 안에 있음

**📌 4. Spark 병렬 처리 + 필터링**

count = spark.sparkContext \

.parallelize(range(1, n + 1), partitions) \

.filter(inside) \

.count()

* [1, 2, ..., n]까지의 숫자를 병렬화하여 RDD로 만듦
* 각 요소마다 inside() 함수를 적용해서 **원 안에 들어간 점의 수를 셈**

**📌 5. π 근사 계산**

print("Pi is roughly %f" % (4.0 \* count / n))

* π ≈ 4 × (원 안에 들어간 점의 수) / (전체 점의 수)
* 몬테카를로 방식으로 추정
* 몬테카를로 방식(Monte Carlo Method)은 수학적 문제를 **확률적 시뮬레이션(무작위 샘플링)** 으로 해결하는 기법입니다.  
  복잡한 계산을 직접 하지 않고, **랜덤한 실험을 많이 수행해서 통계적으로 근사값을 구하는 방법**입니다.

**📘 예: π(파이) 값 추정**

**🎲 실험 설정**

* 단위 정사각형 (1×1)에 무작위로 점을 찍음
* 그 안에 반지름 1의 **원 (사분원)** 을 그려놓음
* (x, y)가 원 안에 들어가는 경우를 셈 (x² + y² < 1)

**📐 면적 비율**

* 정사각형 면적: 1
* 사분원 면적: π/4
* 무작위 점 중 **원 안에 들어간 비율 ≈ π/4**

**💡 추정식**

**폰트, 텍스트, 화이트, 친필이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.**

**🧪 예시**

| **점 개수** | **원 안 점 비율** | **추정된 π** |
| --- | --- | --- |
| **1,000** | **0.78** | **3.12** |
| **100,000** | **0.7855** | **3.142** |
| **1,000,000** | **0.7854** | **3.1416** |

**→ 점을 많이 찍을수록 실제 π 값(3.14159...)에 가까워짐**

**📌 6. Spark 종료**

spark.stop()

**🧠 작동 원리 요약**

1. (0,0)~(1,1) 범위 내에서 무작위 점을 많이 찍고
2. 이 중 원 안에 들어간 점의 비율을 구해서
3. 원의 넓이 비율로 π를 근사 추정

**✅ 실행 결과**

Pi is roughly 3.141280

* 파티션 수가 많고, 반복 횟수가 많을수록 정밀도가 높아집니다.

Airflow DAG log 출력

텍스트, 스크린샷, 소프트웨어, 웹 페이지이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

**Apache Airflow (MWAA)에서 EMR Serverless 작업이 실행된 이력**이며, **start\_job이라는 태스크가 정상적으로 실행되고 성공적으로 완료된 로그**입니다.

**타임라인 흐름과 주요 메시지 분석**

**✅ 전체 요약**

| **항목** | **값** |
| --- | --- |
| DAG ID | example\_emr\_serverless\_job |
| Task ID | start\_job |
| 실행 시각 | 2025-04-05 13:02:39 UTC |
| Application ID | 00frgtbdmdkjhr09 |
| Job ID | 00frgu7dp4gt500b |
| 최종 상태 | ✅ **SUCCESS** (성공적으로 실행됨) |

**🧭 실행 흐름 상세 분석**

| **시간** | **설명** |
| --- | --- |
| **13:02:43** | 모든 의존 조건 충족, 태스크 실행 시작 준비 완료 |
| **13:02:44** | EMR Serverless Job 제출 시작 |
| **13:02:44** | Application ID: 00frgtbdmdkjhr09, Job ID: 00frgu7dp4gt500b 생성됨 |
| **13:03:14 ~ 13:05:15** | Job 상태 확인 Polling 시도 |
| ⤷ Try 1~6 | 상태 변화: SUBMITTED → PENDING → SCHEDULED → RUNNING |
| **13:05:45** | 최종 상태: ✅ SUCCESS 로 완료됨 |
| ⤷ Airflow 태스크가 **성공(SUCCESS)** 로 마크되고 종료됨 (exit code 0) |  |

**🔍 주요 로그 라인 설명**

| **로그 라인** | **의미** |
| --- | --- |
| INFO - Executing <Task(EmrServerlessStartJobOperator)... | EMR Serverless용 오퍼레이터가 실행되었음을 의미 |
| INFO - EMR serverless job started: 00frgu7dp4gt500b | 실제 Job ID 할당됨 |
| INFO - Try x: Query is still in non-terminal state - ... | Job 상태를 지속적으로 모니터링 (Airflow가 polling 중) |
| Final state is SUCCESS | Job 실행 성공 |
| Marking task as SUCCESS | Airflow에서 이 TaskInstance가 성공으로 처리됨 |
| Task exited with return code 0 | 정상 종료 |

**🧠 참고 사항**

* **Job 상태 변화는 정상적인 흐름**입니다:  
  SUBMITTED → PENDING → SCHEDULED → RUNNING → SUCCESS
* 이 태스크는 보통 EmrServerlessStartJobOperator 또는 커스텀 오퍼레이터를 통해 EMR Serverless Job을 시작합니다.
* 로그를 보면 \*\*Connection ID: aws\_default\*\*를 사용했으므로, 해당 연결이 올바르게 IAM 역할과 권한을 갖고 있다는 것도 확인됩니다.

**✅ 지금까지 문제 없음!**

이 로그를 기반으로 확인할 수 있는 것은:

* **작업이 정상적으로 제출되고 실행됨**
* **Airflow Task도 성공(SUCCESS)**
* **EMR Serverless Job도 최종적으로 성공(SUCCESS)**

**Airflow UI DAG**

텍스트, 스크린샷, 소프트웨어, 폰트이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

Airflow UI DAG를 통해 DAG(워크플로)의 상태와 실행 이력 등을 한눈에 확인할 수 있다.

**📋 Airflow UI DAG 리스트 화면 주요 정보**

| **항목** | **설명** |
| --- | --- |
| **DAG ID** | DAG의 고유 식별자 (예: example\_emr\_serverless\_job) |
| **Owner** | DAG 소유자 (보통 airflow, 사용자 지정 가능) |
| **Runs** | 이 DAG이 실행된 총 횟수 (현재 **1회 실행됨**) |
| **Schedule** | DAG 실행 주기 (None: 수동 실행 전용) |
| **Last Run** | 가장 최근 실행된 시각 (2025-04-05, 13:02:39 UTC) |
| **Next Run** | 예약된 다음 실행 시간 (스케줄이 없으므로 비어있음) |
| **Recent Tasks** | 최근 실행된 태스크들의 상태 (녹색 원 → 성공) |
| **Actions** | ▶ 실행 / 🗑 삭제 / ⋮ 기타 상세 보기 (Graph View 등) |

**🟢 아이콘 해석**

* **왼쪽 파란색 스위치**: DAG이 활성 상태인지 여부  
  → 켜져 있으므로, 트리거할 준비가 된 상태
* **녹색 원 (Recent Tasks)**: 최근 실행된 태스크가 **성공(SUCCESS)** 했음을 의미
* **▶ 버튼 (Action)**: DAG을 **수동 실행(Trigger)** 하기 위한 버튼
* **🗑 버튼**: DAG 삭제
* **… (점 3개 메뉴)**: Tree View, Graph View, Code 등 상세 메뉴

**📊 현재 상태 요약**

| **항목** | **상태** |
| --- | --- |
| DAG 활성화 | ✅ (켜져 있음) |
| 스케줄 설정 | ❌ 없음 (None) — 수동 실행용 |
| 실행 횟수 | ✅ 1회 실행됨 |
| 실행 성공 여부 | ✅ 성공적으로 완료됨 (녹색 원) |
| 마지막 실행 시간 | 2025-04-05 13:02:39 UTC |
| 다음 실행 예정 | 없음 (스케줄 없으므로) |

**🧠 유용한 팁**

| **원하는 정보** | **어디서 확인?** |
| --- | --- |
| DAG 흐름 시각화 | ... 메뉴 → **Graph View** |
| 태스크 간의 의존성 보기 | ... 메뉴 → **Tree View** |
| 로그 확인 | DAG 클릭 → 실행 시점 선택 → Task 클릭 → **Log** |
| DAG 코드 확인 | ... 메뉴 → **Code** (Python 원본 DAG 코드) |

**✅ 요약**

Airflow UI DAG 리스트 화면은 다음을 빠르게 확인할 수 있다:

* DAG이 **정상 등록되었는지**
* 마지막 실행이 언제였고, **성공했는지 실패했는지**
* 스케줄러가 돌고 있는지 여부
* DAG을 수동으로 **재실행하거나 삭제할 수 있는 액션 버튼**

**트랜잭션 데이터 레이크**

**아파치 후디**

윈도우에서 Hudi-scripts.zip 압축파일을 다운 받아 풀고 파이썬 파일 4개를

Amazon S3->버킷->emrserverless-workshop-891377038690 아래에 업로드해놓는다

EMR Studio -> Applications에 My\_First\_Application을 새로 만들어 진행한다

job 실행 후 Amazon Athena -> 쿼리 편집기에서 결과를 확인한다

텍스트, 스크린샷, 소프트웨어, 디스플레이이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

**hudi-cow.py 파이썬 소스 분석**

이 PySpark 스크립트는 Hudi(Copy-on-Write, COW) 형식으로 대용량 데이터를 S3에 저장하고, Hive와의 연동 설정까지 포함된 데이터 적재 파이프라인입니다.

**📌 주요 목적**

* PySpark를 사용해 Hudi 포맷으로 데이터를 생성하고, AWS S3에 저장
* Hive와의 메타데이터 싱크 설정 포함
* Hudi의 **Copy-On-Write** 저장 방식 사용

Hudi의 Copy-On-Write (COW)\*\*는 저장 방식 중 하나로, **데이터의 일관성과 쿼리 성능을 우선시**할 때 사용하는 방식이다

**📦 Copy-On-Write (COW)란?**

✅ **업데이트 발생 시 기존 파일을 수정하지 않고, 새로운 파일로 복사(Copy) 후 수정(Update)**  
기존 데이터를 *직접 덮어쓰지 않고*, 업데이트된 내용이 반영된 새로운 버전을 생성하는 방식이다.

**🧠 COW의 특징**

| **항목** | **설명** |
| --- | --- |
| 📂 저장 방식 | 새로운 Parquet 파일 생성 |
| 🔄 업데이트 처리 | 기존 파일은 그대로 두고, 변경된 데이터만 포함된 새 파일 작성 |
| ⚡ 읽기 성능 | **높음** — 항상 최신 데이터가 파일에 존재 |
| 🐢 쓰기 성능 | **낮음** — 업데이트마다 전체 파일을 다시 씀 (병렬 처리로 완화 가능) |
| 🔍 주로 사용 시점 | 실시간 읽기 위주 (BI 툴, 보고서 등) |
| 📈 예시 사용 | 대시보드, 실시간 분석, Presto/Trino 쿼리 |

**🔁 vs. Merge-On-Read (MOR) 비교**

| **항목** | **Copy-On-Write (COW)** | **Merge-On-Read (MOR)** |
| --- | --- | --- |
| 저장 형식 | Parquet | Base 파일(Parquet) + 로그 파일(Avro) |
| 읽기 속도 | 빠름 | 느릴 수 있음 (로그와 병합 필요) |
| 쓰기 속도 | 느림 | 빠름 (로그만 쓰면 됨) |
| 쿼리 최적화 | Snapshot 쿼리에 적합 | Incremental 쿼리 또는 Change Data Capture에 적합 |
| 실사용 예 | 데이터 분석, BI 도구 | 실시간 스트리밍 업데이트, Change Tracking |

**📖 Merge-On-Read (MOR)란?**

"일단 메모장에 적어두고, 나중에 필요할 때 원본이랑 합쳐서 보여줄게!"

**📦 비유로 이해하기**

| **상황** | **설명** |
| --- | --- |
| COW | 매번 책 전체를 다시 인쇄해서 최신 버전 만들기 |
| MOR | 책 원본은 그대로 두고, 수정사항은 포스트잇에 메모 → 나중에 읽을 때 원본 + 포스트잇 합쳐서 보여주기 |

**✅ 정리**

* **MOR = 변경사항은 로그에 저장, 읽을 때 병합**
* **실시간 업데이트가 많고, 읽는 건 나중에 해도 되는 경우에 적합**
* 스트리밍 기반 데이터 적재 또는 변경 이력 추적 등에 많이 사용됨

**📁 예시**

.option("hoodie.datasource.write.operation", "bulk\_insert")

또는

.option("hoodie.datasource.write.operation", "upsert")

이 설정을 사용하는 경우, Copy-On-Write 모드로 동작한다.

**✅ 요약**

* **Copy-On-Write (COW)** = 파일을 새로 만들어서 저장
* **읽기 성능이 뛰어남**, 하지만 쓰기/업데이트 시 자원이 더 많이 듬
* **데이터 정합성과 쿼리 성능이 중요한 경우 추천**

**소스 분석 계속**

**🔧 1. 초기 설정**

import sys

from pyspark.sql import SparkSession

if len(sys.argv) == 1:

print('no arguments passed')

sys.exit()

* S3 버킷 이름을 인자로 받습니다. 인자가 없으면 종료합니다.
* 예시 실행: spark-submit script.py my-bucket-name

spark = SparkSession\

.builder\

.appName("hudi\_cow")\

.getOrCreate()

* Spark 세션 생성

**📁 2. 상수 정의**

**Hudi 관련 설정 (Hudi Write Options)**

* RECORDKEY\_FIELD\_OPT\_KEY: Hudi 내부에서 데이터의 고유 키
* PRECOMBINE\_FIELD\_OPT\_KEY: 최신 데이터를 판단하는 기준 필드
* OPERATION\_OPT\_KEY: 쓰기 모드 (bulk\_insert, upsert, delete)
* PAYLOAD\_CLASS\_OPT\_KEY: 삭제 연산을 위한 EmptyPayload 등

**Hive Sync 관련 설정**

* HIVE\_SYNC\_ENABLED\_OPT\_KEY: Hive 테이블 생성/동기화 여부
* HIVE\_PARTITION\_FIELDS\_OPT\_KEY 등으로 파티셔닝도 설정 가능

**기타 설정**

* 파티션, bootstrap, incremental query 등 Hudi의 다양한 옵션 정의됨 (실제로 사용된 건 일부)

**🧪 3. 샘플 데이터 생성**

def get\_json\_data(start, count, dest):

time\_stamp = datetime.now().strftime('%Y-%m-%d %H:%M:%S')

data = [{"trip\_id": i, "tstamp": time\_stamp, "route\_id": chr(65 + (i % 10)), "destination": dest[i%10]} for i in range(start, start + count)]

return data

* trip\_id: 정수 인덱스
* route\_id: A~J (ASCII 문자)
* destination: 10개 도시 리스트
* 현재 시간 기준 타임스탬프를 가짐

def create\_json\_df(spark, data):

sc = spark.sparkContext

return spark.read.json(sc.parallelize(data))

* Spark의 RDD로 JSON 데이터를 읽어서 DataFrame 생성

**🧾 4. 쓰기 설정**

(df1.write.format(HUDI\_FORMAT)

.option(PRECOMBINE\_FIELD\_OPT\_KEY, config["sort\_key"]) # 최신 데이터 기준

.option(RECORDKEY\_FIELD\_OPT\_KEY, config["primary\_key"]) # 고유 키

.option(TABLE\_NAME, config['table\_name']) # 테이블 이름

.option(OPERATION\_OPT\_KEY, BULK\_INSERT\_OPERATION\_OPT\_VAL) # bulk\_insert 사용

.option(BULK\_INSERT\_PARALLELISM, 10) # 병렬 처리

.option(HIVE\_TABLE\_OPT\_KEY,config['table\_name'])

.option(HIVE\_SYNC\_ENABLED\_OPT\_KEY,"true") # Hive 연동

.mode("Overwrite") # 전체 덮어쓰기

.save(OUTPUT\_BUCKET))

* bulk\_insert: 빠른 초기 적재에 유리하나 업데이트 기능 없음 (vs upsert)
* mode("Overwrite"): S3 위치에 덮어쓰기
* Hive Sync: 테이블 자동 생성 및 메타스토어 등록 (hive-site.xml 사전 구성 필요)

**📍 결과**

* s3://<bucket-name>/hudi/hudi\_trips\_cow 위치에 Hudi 테이블 생성
* Hudi 테이블은 COW 방식으로 저장
* Hive에 해당 테이블 자동 등록됨 (Glue 메타스토어나 Hive 메타스토어 사용 가능)

**🔁 COW vs MOR 비교 정리**

| **항목** | **Copy-On-Write (COW)** | **Merge-On-Read (MOR)** |
| --- | --- | --- |
| 쓰기 성능 | 느림 | 빠름 ✅ |
| 읽기 성능 | 빠름 ✅ | 느릴 수 있음 |
| 병합 시점 | 쓰기 시점 | 읽기 시점 |
| 사용 시나리오 | 대시보드, 분석 | 실시간 업데이트, 로그 기록 등 |

**bulk\_insert와 upsert의 차이**

Hudi에서 데이터를 쓰는 방식(write operation)인데 역할이 다르고 성능에도 차이가 있다

**📌 핵심 요약**

| **항목** | **bulk\_insert** | **upsert** |
| --- | --- | --- |
| 목적 | 빠른 초기 적재 (처음 넣을 때) | 데이터 삽입 + 수정 (계속 갱신) |
| 중복 키 처리 | 무시 (같은 키가 여러 개 있어도 마지막만 남을 수도 있음) | 기존 데이터와 병합 |
| Precombine 적용 | ❌ 적용 안 됨 (정렬도 안 함) | ✅ 최신 레코드만 남도록 정렬 |
| 성능 | ⚡ 매우 빠름 | 🐢 상대적으로 느림 |
| 사용 시점 | **초기 적재, 마이그레이션** | **정기 업데이트, 스트리밍** |
| 지원 모드 | COW / MOR | COW / MOR |
| 레코드 충돌 처리 | 없음 | 자동으로 최신 레코드만 유지 |

**📦 bulk\_insert 특징**

* **초기 적재용**
* 병렬로 빠르게 파일을 만들기 때문에 속도가 빠름
* 중복 제거, 정렬, 병합 처리 없음 → 오로지 "쓰기"만
* 주로 Hive 테이블이나 S3에 **최초로 데이터를 밀어넣을 때** 사용

🔧 예시:

.option("hoodie.datasource.write.operation", "bulk\_insert")

**🔁 upsert 특징**

* 삽입 + 업데이트 동시 처리
* **기존 키가 있으면 덮어쓰고**, 없으면 새로 추가
* recordkey와 precombine key를 기준으로 최신 데이터 판별
* CDC(Change Data Capture) 또는 스트리밍/배치 업데이트에서 사용

🔧 예시:

.option("hoodie.datasource.write.operation", "upsert")

**🧪 실전 예시 상황**

| **상황** | **어떤 걸 써야 할까?** |
| --- | --- |
| 데이터 1억 건 초기 적재 | ✅ bulk\_insert |
| 매일 변경된 데이터 추가 | ✅ upsert |
| 주기적인 전체 데이터 덮어쓰기 | 가능은 하나 bulk\_insert + Overwrite 추천 |
| Kafka 스트리밍 수집 | ✅ upsert (또는 MOR + streaming write) |

**💡 주의할 점**

* bulk\_insert는 정말 빠르지만, **기존 데이터를 덮어쓰지 않으므로 중복 관리 안 됨**
* upsert는 중복 키와 병합 처리가 있으므로 **속도는 느리지만 안정성은 높음**
* 두 방식은 Hudi 내부 레코드 관리 방식에 영향을 미치므로 **혼용 주의**

upsert는 데이터베이스 용어에서 나온 말로, 두 단어의 합성어입니다:

**upsert = update + insert**

**🔤 구성**

* **Update**: 기존에 같은 키(Primary Key)가 있으면 → 값을 수정
* **Insert**: 같은 키가 없으면 → 새로 삽입

즉,

**"있으면 수정하고, 없으면 삽입하라"**  
라는 동작을 한 번에 수행하는 연산이다.

**💡 예를 들면?**

**현재 데이터:**

| **id** | **name** |
| --- | --- |
| 1 | Alice |
| 2 | Bob |

**새로운 데이터:**

| **id** | **name** |
| --- | --- |
| 2 | Robert |
| 3 | Charlie |

**upsert 후 결과:**

| **id** | **name** |
| --- | --- |
| 1 | Alice |
| 2 | Robert |
| 3 | Charlie |

**🔧 Hudi에서의 upsert**

Hudi에서는 recordkey 기준으로 기존 데이터를 찾아서,

* 최신이면 덮어씀 (precombine field 기준 비교)
* 아니면 무시
* 없으면 새로 추가

이렇게 처리한다.

**hudi-upsert-cow.py 소스 분석**

|  |
| --- |
| UPSERT\_OPERATION\_OPT\_VAL = "upsert"  DELETE\_OPERATION\_OPT\_VAL = "delete"  …  upsert\_dest = ["Boston", "Boston", "Boston", "Boston", "Boston","Boston","Boston","Boston","Boston","Boston"]  df = create\_json\_df(spark, get\_json\_data(1000000, 10, upsert\_dest))  …  (df.write.format(HUDI\_FORMAT)  …  .option(OPERATION\_OPT\_KEY, UPSERT\_OPERATION\_OPT\_VAL)  … |

이 소스는 trip\_id 1000000~1000009 까지 10개의 destination을 Boston으로 수정하는 코드이다

**hudi-mor.py 소스 분석**

|  |
| --- |
| (df1.write.format(HUDI\_FORMAT)  .option(PRECOMBINE\_FIELD\_OPT\_KEY, config["sort\_key"])  .option(RECORDKEY\_FIELD\_OPT\_KEY, config["primary\_key"])  .option(TABLE\_NAME, config['table\_name'])  .option(OPERATION\_OPT\_KEY, BULK\_INSERT\_OPERATION\_OPT\_VAL)  .option(BULK\_INSERT\_PARALLELISM, 10)  .option(HIVE\_TABLE\_OPT\_KEY,config['table\_name'])  .option(HIVE\_SYNC\_ENABLED\_OPT\_KEY,"true")  .option(STORAGE\_TYPE\_OPT\_KEY, "MERGE\_ON\_READ")  .mode("Overwrite")  .save(OUTPUT\_BUCKET)) |

다음 줄이 핵심이다:

.option(STORAGE\_TYPE\_OPT\_KEY, "MERGE\_ON\_READ")

즉,

.option("hoodie.datasource.write.storage.type", "MERGE\_ON\_READ")

이 옵션이 저장 방식을 MOR로 설정한다.  
기본값은 "COPY\_ON\_WRITE"이므로, 명시적으로 "MERGE\_ON\_READ"로 바꿔줘야 MOR이 된다.

Athena에서 아래 명령은 두 번 따로 실행하여 결과를 비교해본다 결과가 동일하다

select count(\*) from default.hudi\_trips\_mor\_ro

select count(\*) from default.hudi\_trips\_mor\_rt

결과가 동일하다

select trip\_id, route\_id, tstamp, destination from default.hudi\_trips\_mor\_ro where trip\_id between 999996 and 1000013

select trip\_id, route\_id, tstamp, destination from default.hudi\_trips\_mor\_rt where trip\_id between 999996 and 1000013

결과가 동일하다

hudi\_trips\_mor\_ro와 hudi\_trips\_mor\_rt는 Hudi MOR(Merge-On-Read) 테이블의 두 가지 뷰(Read-Optimized와 Real-Time)인데, **결과가 동일할 수도 있고, 다를 수도 있다**. 현재 동일하게 나오는 이유는 **특정 조건**이 만족되었기 때문이다.

**🧠 이유: compaction 전 or 아직 log 파일이 없음**

**1. 초기 bulk\_insert만 수행된 상태**

* 현재 코드에서는 다음과 같이 only bulk\_insert 작업만 실행되었다:

.option(OPERATION\_OPT\_KEY, BULK\_INSERT\_OPERATION\_OPT\_VAL)

.option(STORAGE\_TYPE\_OPT\_KEY, "MERGE\_ON\_READ")

* 이 경우, **기본적으로 모든 데이터는 base 파일에 저장**되고, 아직 delta log 파일이 생성되지 않았다.

**✅ 결과적으로:**

* \_ro: base 파일 읽음 → 전체 데이터 조회 가능
* \_rt: base + log (현재 없음) 병합 → 결과 동일

**✅ 컴팩션(compaction)이란?**

MOR 테이블에서 delta log 파일에 기록된 변경 사항(insert/update/delete)을 base 파일에 반영(parquet 파일로 병합)하는 작업이다.

**💥 그런데 왜 컴팩션이 필요할까?**

| **이유** | **설명** |
| --- | --- |
| 성능 향상 | log 파일이 계속 쌓이면 read 성능 저하 → 병합해서 새 base 파일 생성 |
| 데이터 정리 | 오래된 delta log를 base 파일에 반영하고 삭제 가능 |
| read-optimized 쿼리 지원 | \_ro 뷰는 log 파일을 못 읽기 때문에, 최신 데이터를 \_ro에서도 보기 위해선 컴팩션 필요 |

**2. log 파일이 존재하더라도 base file과 병합 후 결과가 같을 수도 있음**

* log 파일에 들어 있는 데이터가 기존 base 파일과 **중복되지 않거나**
* log에 있는 업데이트가 base에 **이미 반영된 형태**로 compaction된 경우

**🔍 확인 포인트**

**✔ 확인 1: log 파일 유무 확인**

* S3 경로 확인: hudi\_trips\_mor/.hoodie/ 또는 .../.log.\* 파일이 있는지 확인해본다.
* log 파일이 없다면 당연히 \_ro와 \_rt는 동일한 결과를 보여줍니다.

**✔ 확인 2: compaction 여부**

* compaction이 자동 또는 수동으로 실행되지 않았으면 log 파일에 있는 변경 사항은 \_ro에 보이지 않음.
* compaction 이후엔 \_ro에 반영될 수 있음.

**✅ 정리**

| **조건** | **\_ro vs \_rt 결과** |
| --- | --- |
| bulk\_insert만 수행 (log 없음) | ✅ 동일 |
| log에만 존재하는 데이터 있음 | ❌ 달라짐 |
| compaction으로 base에 반영됨 | ✅ 다시 동일 가능 |

select count(\*) from default.hudi\_trips\_mor\_ro 명령 실행시

텍스트, 폰트, 스크린샷, 라인이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

으로 나오고

select count(\*) from default.hudi\_trips\_mor\_rt 명령수행시



으로 나오는 이유는?

**✅ 이유 분석**

**✅ 1. \_ro: Parquet 파일만 읽음 → 빠름**

* \_ro는 \*\*base file (Parquet 파일)\*\*만 읽기 때문에 Spark 또는 Athena가 \*\*단순 컬럼 통계(metadata)\*\*만으로 count(\*)를 계산할 수 있다.
* 따라서, **실제 데이터를 로드하지 않아도 됨 → "스캔한 데이터: - "** 로 표시됨.
* count(\*)처럼 단순한 집계는 parquet의 footer metadata에 row count 정보가 있으므로 빠르게 처리 가능.

**✅ 2. \_rt: base + log 병합 → 느림**

* \_rt는 \*\*log 파일(Hudi delta log)\*\*까지 읽고 \*\*기존 parquet과 병합(Merge)\*\*해야 하므로, 실제로 데이터를 스캔하고, row를 카운트해야 한다.
* Athena는 Hudi log 파일을 병합하기 위해 내부적으로 Java worker에서 처리 작업을 실행 → **실제 I/O + 병합 비용** 발생
* 이로 인해 **스캔된 데이터도 존재**하고 **실행 시간도 길어짐 (4.5초)**

**💡 기술적으로 어떤 일이 일어나나?**

| **뷰** | **내부 동작** |
| --- | --- |
| \_ro | count(\*) → parquet metadata에서 row count만 조회 |
| \_rt | count(\*) → parquet + log 파일을 모두 읽고 병합한 후 row 수 계산 |

**📌 참고: 실제 로우 수가 바뀌지 않았더라도 \_rt는 항상 전체 병합**

* even if upsert가 없어도, \_rt는 "항상 최신 보기"이므로 병합 과정을 생략하지 않는다.
* \_ro는 compaction이 되지 않은 상태에서는 최신 데이터를 반영하지 않지만, 성능은 매우 빠르다.

**🧠 정리**

| **항목** | **\_ro** | **\_rt** |
| --- | --- | --- |
| 병합 처리 | ❌ 없음 | ✅ base + log 병합 필요 |
| parquet footer 사용 | ✅ 사용함 | ❌ 사용 불가 |
| 성능 | ✅ 매우 빠름 | ❌ 상대적으로 느림 |
| 스캔한 데이터 | 거의 없음 (메타데이터 기반) | 실제 파일 읽음 |
| 정확도 | compaction 이전이면 누락 가능 | 항상 최신 상태 |

**hudi-upsert-mor.py 소스 분석**

|  |
| --- |
| …  (df.write.format(HUDI\_FORMAT)  .option(PRECOMBINE\_FIELD\_OPT\_KEY, config["sort\_key"])  .option(RECORDKEY\_FIELD\_OPT\_KEY, config["primary\_key"])  .option(TABLE\_NAME, config['table\_name'])  .option(OPERATION\_OPT\_KEY, UPSERT\_OPERATION\_OPT\_VAL)  .option(UPSERT\_PARALLELISM, 10)  .option(HIVE\_TABLE\_OPT\_KEY,config['table\_name'])  .option("hoodie.compact.inline.max.delta.commits",2)  .option(HIVE\_SYNC\_ENABLED\_OPT\_KEY,"true")  .option(STORAGE\_TYPE\_OPT\_KEY, "MERGE\_ON\_READ")  .mode("Append")  .save(OUTPUT\_BUCKET)) |

**Apache Hudi의 Merge-On-Read(MOR) 테이블에 대해 Upsert 작업을 수행하는 코드이다.  
MOR 방식 + Upsert 연산임을 나타내는 부분은 아래와 같다.**

✅ 1. MOR(Merge-On-Read) 방식임을 나타내는 부분

.option(STORAGE\_TYPE\_OPT\_KEY, "MERGE\_ON\_READ")

* 실제 값은 다음과 같다:

.option("hoodie.datasource.write.storage.type", "MERGE\_ON\_READ")

* Hudi 테이블의 저장 방식이 Merge-On-Read(MOR) 방식임을 지정하는 핵심 설정이다.
* 이 옵션이 없다면 기본값은 Copy-On-Write(COW) 방식이다.

✅ 2. Upsert 연산임을 나타내는 부분

.option(OPERATION\_OPT\_KEY, UPSERT\_OPERATION\_OPT\_VAL)

* 실제 값은 다음과 같다:

.option("hoodie.datasource.write.operation", "upsert")

* Hudi는 insert, upsert, bulk\_insert, delete 등의 연산을 지원하며, 여기서는 기존 데이터를 키 기준으로 덮어쓰는 upsert 연산을 수행하고 있다.

✅ 보조적 근거

다음 설정은 MOR 환경에서 컴팩션을 유도하여 \_ro 뷰에서도 데이터가 최신화될 수 있도록 지원한다:

.option("hoodie.compact.inline.max.delta.commits", 2)

* 2번 커밋 후 자동(compaction inline)으로 병합이 수행되도록 설정한 것이다.
* MOR 테이블에서 \*\*log 파일이 base 파일로 병합(compaction)\*\*되는 기준이 된다.

✅ 요약

| 코드 | 설명 |
| --- | --- |
| .option("hoodie.datasource.write.storage.type", "MERGE\_ON\_READ") | MOR 방식 지정 |
| .option("hoodie.datasource.write.operation", "upsert") | Upsert 연산 지정 |
| .option("hoodie.compact.inline.max.delta.commits", 2) | 인라인 컴팩션 설정 (보조적 설정) |

이 세 가지가 이 코드가 MOR 테이블에 대한 Upsert 연산을 수행한다는 명확한 증거이다.

**🔍 \_ro(Read Optimized) 와 \_rt(Real Time) 테이블 비교 요약**

| **테이블 접미어** | **의미** | **설명** |
| --- | --- | --- |
| \_ro | **Read Optimized** | base file(parquet)만 읽음 → 빠름, 최신 변경사항 누락 가능 |
| \_rt | **Real Time** | base file + log file 병합 → 느릴 수 있음, 최신 데이터 제공 |

**✅ 비교 표**

| **항목** | **\_ro (Read Optimized)** | **\_rt (Real Time)** |
| --- | --- | --- |
| **데이터 소스** | base file (parquet) | base file + delta log (Hudi log file) 병합 |
| **최신 데이터 반영 여부** | ❌ compaction 이전 변경사항은 반영 안됨 | ✅ 최신 데이터 포함 (업서트/삭제 즉시 반영) |
| **성능** | ✅ 빠름 (병합 작업 없음) | ❌ 느릴 수 있음 (실시간 병합 필요) |
| **사용 목적** | 리포트, BI, 배치 쿼리 등 정적인 분석에 적합 | 실시간 조회, CDC 파이프라인 등에 적합 |
| **Athena 지원 여부** | ✅ 지원 | ✅ 지원 (단, 복잡한 쿼리는 느릴 수 있음) |
| **compaction 영향** | compaction 후에야 최신 데이터가 반영됨 | compaction 여부 상관없이 항상 최신 데이터 제공 |
| **데이터 정합성** | 약간 느린 consistency (compaction 의존) | 높은 consistency (즉시 변경 반영) |

**✅ 예제**

**\_ro 쿼리:**

SELECT \* FROM default.hudi\_trips\_mor\_ro WHERE trip\_id = 1000000;

* 최신 데이터가 반영되지 않았을 수 있음 (compaction이 안 되었을 경우)

**\_rt 쿼리:**

SELECT \* FROM default.hudi\_trips\_mor\_rt WHERE trip\_id = 1000000;

* 업서트 또는 삭제 직후의 상태가 반영됨

**🔁 언제 어떤 뷰를 써야 할까?**

| **상황** | **추천 뷰** |
| --- | --- |
| 성능이 중요하고, 약간 지연된 데이터 허용 | \_ro |
| 실시간 분석 또는 변경사항 추적이 중요 | \_rt |
| Athena로 정기 배치 리포트 작성 | \_ro |
| Kafka Connect 또는 Flink CDC 처리 | \_rt |

**🧠 마무리 요약**

* \_ro: 빠르지만 **log 파일 미반영**, compaction 필요
* \_rt: 느릴 수 있지만 **항상 최신 상태**, 실시간 분석에 적합

**\_ro(Read Optimized) 와 \_rt(Real Time) 테이블 성능 비교**

select trip\_id, route\_id, tstamp, destination from default.hudi\_trips\_mor\_ro where trip\_id between 999996 and 1000013



select trip\_id, route\_id, tstamp, destination from default.hudi\_trips\_mor\_rt where trip\_id between 999996 and 1000013



**✅ 쿼리 성능 차이 비교**

| **항목** | **\_ro 뷰** | **\_rt 뷰** |
| --- | --- | --- |
| 실행 시간 | **0.9초** | **4.0초** |
| 스캔한 데이터 | **10.28KB** | **22.40MB** |
| 대기열 시간 | 75ms | 145ms |

**🔍 왜 이런 차이가 생길까?**

**🔸 \_ro 뷰 (Read Optimized)**

* **Parquet base 파일만 읽음**
* log 파일에 있는 업데이트 정보는 **무시**
* 따라서 읽어야 할 파일이 작고, **I/O도 적고 빠름**
* 실질적으로 **10KB 수준의 작은 파일만 스캔**

**🔸 \_rt 뷰 (Real Time)**

* **base 파일 + log 파일 병합**
* log 파일은 최신 업데이트 내용을 포함
* Athena가 log 파일을 읽고, base와 \*\*실시간으로 병합(merge)\*\*해야 함
* 이 병합 과정에서 **데이터 양이 커지고, 처리 시간도 길어짐**
* 스캔 데이터가 22MB까지 증가

**🧠 핵심 요약**

| **항목** | **\_ro** | **\_rt** |
| --- | --- | --- |
| 데이터 읽기 방식 | Parquet만 읽음 | Parquet + log 병합 |
| 최신 데이터 포함 여부 | ❌ compaction 전에는 안 보임 | ✅ 항상 최신 |
| 처리 속도 | 빠름 | 느림 |
| I/O 비용 | 낮음 (10KB) | 높음 (22MB) |
| 쿼리 목적 | 정적 분석, 리포트 | 실시간 분석, 최신 변경 확인 |

**💡 쉽게 비유하자면**

| **비교 항목** | **예시** |
| --- | --- |
| \_ro 뷰 | 완성된 책을 그냥 읽는 것 |
| \_rt 뷰 | 책 본문 + 수정 포스트잇을 읽으면서 실시간으로 반영하는 것 |

**📌 결론**

* **\_ro는 빠르지만 log에 있는 최신 변경사항은 포함되지 않음**
* **\_rt는 느리지만 항상 최신 상태를 보장**
* 쿼리 목적에 따라 적절히 선택해야 한다

**아파치 아이스버그**

iceberg-create-table job을 실행하고 나서 다음 iceberg-insert-table job을 실행할 때는

EMR Studio의 Batch job runs에서 iceberg-create-table을 선택 체크하고 [Clone]버튼을

클릭하면 다른 설정 값들을 입력하지 않아도 된다 (Name과 Script location만 변경하면 된다)

**iceberg\_create\_table.py 소스분석**

이 PySpark 스크립트는 **EMR Serverless 환경에서 Iceberg 테이블을 생성**하는 예제이다.

**✅ 핵심 목적**

**AWS Glue Catalog + Apache Iceberg + Spark SQL**을 사용해  
tripsdb라는 데이터베이스에 trips라는 Iceberg 테이블을 생성하는 작업이다.

**✅ 핵심 구성 요소 설명**

**1. Spark 세션 생성 + Hive/Glue Catalog 활성화**

spark = SparkSession.builder \

.appName("EMR-Serverless-Iceberg-Demo-create-table") \

.enableHiveSupport() \

.getOrCreate()

* EMR Serverless에서 Hive 및 Glue Catalog 사용을 위한 세션 설정

**2. Glue Catalog + 스키마 이름 지정**

catalog\_name = "glue\_iceberg\_catalog\_demo"

schema\_name = "tripsdb"

table\_name\_1 = "trips"

* Glue에 등록된 Iceberg 카탈로그를 사용
* tripsdb라는 데이터베이스에 테이블 생성 예정

**3. 스키마 생성 및 namespace 설정**

spark.sql("""CREATE SCHEMA IF NOT EXISTS tripsdb""")

spark.sql("use tripsdb")

* Iceberg가 사용할 데이터베이스(tripsdb)를 생성하고 사용 대상으로 설정

**4. Iceberg 테이블 생성**

spark.sql("""

CREATE TABLE glue\_iceberg\_catalog\_demo.tripsdb.trips (

trip\_id int,

tstamp timestamp,

route\_id string,

destination string

)

USING iceberg

PARTITIONED BY (days(tstamp))

""")

* Iceberg 형식으로 테이블 생성
* **days(tstamp)** 기준으로 파티셔닝
* Glue 메타카탈로그에 등록됨

**5. 테이블 존재 확인**

spark.sql("""show tables in tripsdb""").show()

* tripsdb 안에 trips 테이블 생성되었는지 확인

**🧠 핵심 요약**

| **항목** | **설명** |
| --- | --- |
| Spark 환경 | EMR Serverless |
| 카탈로그 | Glue 기반 Iceberg Catalog |
| DB/스키마 | tripsdb |
| 테이블 | trips (Iceberg 형식, 날짜 파티셔닝) |
| 사용 기술 | Spark SQL + Iceberg + Glue Catalog |

이 코드는 **Iceberg 테이블 초기 생성 예제**로, 이후에 데이터를 insert하거나 시간여행(query-as-of-time), snapshot, rollback 등을 할 수 있는 기반이 된다.

**iceberg\_insert\_table.py 소스 분석**

이 소스의 핵심은 **Iceberg 테이블에 두 번에 걸쳐 데이터를 삽입하고**, **스냅샷 ID를 조회**하는 것이다. **데이터 삽입 및 버전 추적(스냅샷) 부분만 설명**하면 다음과 같다:

**✅ 핵심 기능 요약**

**1. 외부 Python 모듈 로딩**

sc.addPyFile("s3://{}/{}/{}".format(s3\_bucket,script\_prefix,demogen\_file))

from demoGenData import Producer

* S3에 있는 demoGenData.py를 로딩하여 동적 코드 실행
* Producer 모듈로부터 데이터 생성 및 DataFrame 생성 함수를 사용

**2. 데이터 생성 및 첫 번째 삽입**

df = Producer.create\_json\_df(spark, Producer.get\_json\_data(0, 1500000, dest))

df.createOrReplaceTempView("tripTempView")

spark.sql("""

INSERT INTO glue\_iceberg\_catalog\_demo.tripsdb.trips

SELECT trip\_id, to\_timestamp(tstamp,'yyyy-MM-dd HH:mm:ss') AS timestamp, route\_id, destination

FROM tripTempView

""")

* trip\_id 0부터 1,500,000개 생성 후 Iceberg 테이블에 삽입

**3. Iceberg 스냅샷 조회 (버전 확인)**

spark.sql("SELECT \* FROM glue\_iceberg\_catalog\_demo.tripsdb.trips.snapshots").show()

* Iceberg 테이블의 **스냅샷 메타데이터**를 조회
* 삽입 전후의 버전 추적 가능

**4. 추가 데이터 삽입 및 스냅샷 비교**

df1 = Producer.create\_json\_df(spark, Producer.get\_json\_data(1500000, 500000, dest))

df1.createOrReplaceTempView("tripTempView2")

spark.sql("""

INSERT INTO glue\_iceberg\_catalog\_demo.tripsdb.trips

SELECT trip\_id, to\_timestamp(tstamp,'yyyy-MM-dd HH:mm:ss') AS timestamp, route\_id, destination

FROM tripTempView2

""")

* 이후 trip\_id 1,500,000부터 500,000개 추가 삽입
* 다시 snapshots 테이블 조회 → **두 개의 스냅샷 존재**

**🧠 핵심 요약**

| **기능** | **설명** |
| --- | --- |
| addPyFile | 외부 Python 데이터 생성기 로딩 |
| Producer.get\_json\_data() | Iceberg 삽입용 JSON 데이터 생성 |
| INSERT INTO ... SELECT ... | Iceberg 테이블에 데이터 삽입 (2번) |
| SELECT FROM snapshots | Iceberg의 버전(스냅샷) 상태 확인 |

이 소스는 **Iceberg의 버저닝 기능을 확인하는 실습 예제**로 적합하며,  
이후 **시간여행(time travel)**, **스냅샷 롤백**, **incremental query**로 확장할 수 있는 기반이 된다.

**iceberg\_schema\_evolution.py 소스 분석**

이 소스의 핵심은 **이미 생성된 Iceberg 테이블에 새로운 컬럼(origin)을 추가**하고,  
**변경된 스키마를 확인**하는 작업이다.

**✅ 핵심 기능 요약**

**1. 현재 카탈로그/스키마/테이블 상태 확인**

spark.sql("use {}".format(catalog\_name)).show()

spark.sql("use {}".format(schema\_name)).show()

spark.sql("show tables in {}".format(schema\_name)).show()

* 현재 Iceberg 카탈로그(glue\_iceberg\_catalog\_demo)와 스키마(tripsdb)를 설정
* 스키마 안의 테이블 목록 확인 (trips 존재 여부 확인용)

**2. Iceberg 테이블에 컬럼 추가**

spark.sql("""

ALTER TABLE glue\_iceberg\_catalog\_demo.tripsdb.trips

ADD COLUMNS (

origin string

)

""")

* Iceberg 테이블은 **schema evolution**을 지원하므로 ALTER TABLE ... ADD COLUMNS 구문을 통해 안전하게 컬럼을 추가할 수 있다.
* 기존 데이터에는 origin 컬럼이 없으므로 null로 채워진다.

**3. 변경된 테이블 스키마 확인**

spark.sql("desc glue\_iceberg\_catalog\_demo.tripsdb.trips").show()

* 컬럼 리스트를 조회하여 origin 컬럼이 추가되었는지 확인

**🧠 핵심 요약**

| **기능** | **설명** |
| --- | --- |
| ALTER TABLE ... ADD COLUMNS | Iceberg 테이블에 **컬럼 동적 추가** |
| DESC 테이블 | 변경된 스키마 확인 |
| Iceberg 특징 | schema evolution 지원 (컬럼 추가, 순서 유지됨) |

이 코드는 **Iceberg의 schema evolution 기능 시연 예제**로,  
추가된 컬럼에 대해 이후 **insert, update, select** 실습으로 확장할 수 있다.  
예를 들어 origin 값을 포함한 insert 테스트나 null 채워진 행 조회가 그 예이다.

**iceberg\_update\_table.py 소스 분석**

이 소스의 핵심은 **Iceberg 테이블에 대해 SQL UPDATE 문을 실행하여 특정 조건을 만족하는 행의 컬럼 값을 수정**하는 것이다..

**✅ 핵심 기능 요약**

**1. 업데이트할 DataFrame 생성**

df\_updates = Producer.create\_json\_df\_update(

spark,

Producer.get\_json\_data\_update(0, 2000000, dest, update\_origin)

)

* trip\_id, route\_id, destination, origin 등의 필드가 포함된 업데이트용 데이터 생성
* 이 DataFrame은 Iceberg 테이블에 직접 쓰지 않고 **UPDATE 테스트를 위한 참조 용도**로 사용된다

**2. 특정 조건 필터링해서 데이터 확인**

df\_updates.filter(df\_updates["route\_id"]=="H").show(truncate=False)

* 예시: route\_id = 'H'인 레코드만 확인하여 **업데이트 타겟 행의 구조를 파악**

**3. Iceberg 테이블에서 UPDATE 수행**

spark.sql("""

UPDATE glue\_iceberg\_catalog\_demo.tripsdb.trips

SET origin = 'Baltimore'

WHERE route\_id = 'H'

""")

* Iceberg는 **DML 지원**이 가능하므로, Spark SQL로 직접 UPDATE 실행 가능
* 조건: route\_id = 'H'인 모든 레코드의 origin 값을 'Baltimore'로 변경
* 내부적으로는 Iceberg가 **Copy-on-Write 방식**으로 파일을 새로 작성하며 처리함

**4. 업데이트된 결과 확인**

spark.sql("""

SELECT \* FROM glue\_iceberg\_catalog\_demo.tripsdb.trips

WHERE route\_id = 'H'

LIMIT 25

""").show()

* 업데이트된 행을 조회하여 변경 사항이 반영되었는지 확인

**🧠 핵심 요약**

| **기능** | **설명** |
| --- | --- |
| Iceberg SQL UPDATE | 조건(route\_id = 'H')을 만족하는 행의 origin 값을 일괄 수정 |
| Spark SQL 기반 | UPDATE ... SET ... WHERE ... 문법으로 실행 |
| 변경 확인 | SELECT ... WHERE로 결과 확인 |
| Iceberg 지원 | DML 지원 가능 (UPDATE, DELETE, MERGE 등) |

이 코드는 **Iceberg의 DML 처리 기능 실습 예제**로 , 이후 MERGE, DELETE, rollback 등으로 확장할 수 있다.

**iceberg\_upsert\_table.py 소스 분석**

이 소스의 핵심은 Apache Iceberg 테이블에 대해 MERGE INTO 구문을 사용하여 Upsert(삽입 또는 갱신)을 수행하는 것이다.

**✅ 핵심 기능 요약**

**1. 업서트용 데이터프레임 생성**

df\_upserts = Producer.create\_json\_df\_update(

spark, Producer.get\_json\_data\_update(0, 2000020, dest, upsert\_origin)

)

* trip\_id, tstamp, route\_id, destination, origin을 포함한 200만 건의 업서트용 데이터 생성

**2. 현재 Iceberg 테이블 상태 조회**

df\_current = spark.sql("SELECT \* FROM glue\_iceberg\_catalog\_demo.tripsdb.trips")

df\_current.count()

df\_current.show()

* 기존 Iceberg 테이블의 상태, 레코드 수 및 스키마를 조회하여 변경 전 상태 확인

**3. 업서트 대상 임시 뷰 생성**

df\_check.createOrReplaceTempView("upsertsView")

* MERGE 대상이 되는 데이터셋을 SQL 뷰로 등록

**4. MERGE INTO (UPSERT) 실행**

MERGE INTO glue\_iceberg\_catalog\_demo.tripsdb.trips target

USING upsertsView source

ON target.trip\_id = source.trip\_id

WHEN MATCHED THEN

UPDATE SET ...

WHEN NOT MATCHED THEN

INSERT \*

* trip\_id 기준으로 매칭
* 매칭되면 업데이트, 없으면 새로 삽입 → 즉 **UPSERT**
* Iceberg는 MERGE INTO 문을 통해 Copy-on-Write 기반으로 이 연산을 처리한다

**5. 업서트 후 결과 확인**

df\_new = spark.sql("SELECT \* FROM glue\_iceberg\_catalog\_demo.tripsdb.trips")

df\_new.count()

df\_new.orderBy(df\_new["trip\_id"].desc()).show(50, truncate=False)

* 업서트가 반영된 테이블을 다시 조회
* trip\_id 내림차순으로 50개 출력하여 삽입 + 갱신 데이터 샘플 확인

**🧠 핵심 요약**

| **기능** | **설명** |
| --- | --- |
| MERGE INTO | Iceberg 테이블에 대해 upsert 수행 |
| WHEN MATCHED THEN UPDATE | 기존 레코드 갱신 |
| WHEN NOT MATCHED THEN INSERT | 새 레코드 삽입 |
| 사용 기술 | Spark SQL 기반 Iceberg DML (MERGE INTO) |

이 코드는 Iceberg의 **MERGE INTO 기반 UPSERT 기능 실습 예제**로 적절하며,  
성공 후 스냅샷 조회나 시간여행 기능으로도 확장 가능하다.

**✅ MERGE INTO에서 업데이트와 인서트가 발생하는 조건 요약**

**🔹 기준: trip\_id**

**✅ 1. 업데이트(UPDATE)**

**조건:**

* 새로운 데이터의 trip\_id가 **기존 테이블에 이미 있을 때**

**결과:**

* 해당 행의 값이 덮어쓰기(갱신)\*된다.

**✅ 2. 인서트(INSERT)**

**조건:**

* 새로운 데이터의 trip\_id가 **기존 테이블에 없을 때**

**결과:**

* 새로운 행으로 추가(삽입)된다.

**✅ 기존 테이블이란?**

* 이전에 만들어진 Iceberg 테이블 (trips)
* 현재 S3와 Glue에 저장되어 있는 테이블
* MERGE 대상 테이블이며, 이미 데이터가 들어있는 상태

**🧠 아주 간단한 정리**

| **상황** | **동작** |
| --- | --- |
| trip\_id가 이미 있으면 | ✅ 업데이트 |
| trip\_id가 없으면 | ✅ 인서트 |

즉, MERGE 구문은 **같으면 덮어쓰기, 없으면 새로 추가**하는 업서트를 실행하는 것이다.

**✅ 정리**

| **용어** | **의미** |
| --- | --- |
| 기존 테이블 | Iceberg 포맷으로 이미 존재하는 테이블 (target) |
| 소스 데이터 | 지금 생성해서 임시 뷰로 만든 새로운 데이터 (source) |
| 비교 기준 | trip\_id (기본 키 역할) |

**iceberg\_delete\_table.py 소스 분석**

이 소스는 **Apache Iceberg 테이블에서 특정 조건을 만족하는 데이터를 삭제**하고,  
그 삭제가 **스냅샷과 메타데이터에 어떻게 기록되는지 확인하는 작업**을 수행한다.

**✅ 분석 요약**

**1. 삭제 대상 데이터 조회**

delete\_dest = "New Jersey"

df\_nj = spark.sql("SELECT trip\_id FROM ... WHERE destination = 'New Jersey'")

count\_current = df\_nj.count()

* destination = 'New Jersey'인 레코드를 조회하고 개수를 카운트함
* 삭제 전 상태 확인

**2. DELETE 실행**

spark.sql("DELETE FROM ... WHERE destination = 'New Jersey'")

* Iceberg 테이블에서 해당 조건을 만족하는 데이터를 **전부 삭제**
* Iceberg는 내부적으로 **삭제된 레코드를 새로운 snapshot으로 관리**하며, 실제 데이터를 바로 지우지 않는다

**3. 삭제 후 상태 확인**

df\_nj\_delete = spark.sql("SELECT trip\_id FROM ... WHERE destination = 'New Jersey'")

count\_after\_delete = df\_nj\_delete.count()

* 삭제 후 같은 조건으로 데이터를 다시 조회하여 **0건인지 확인**

**4. 스냅샷 메타데이터 확인**

spark.sql("SELECT \* FROM ...snapshots").show()

* 삭제가 하나의 새로운 snapshot으로 기록되었는지 확인
* 각 snapshot에는 timestamp, operation type(DELETE), snapshot\_id 등이 포함된다

**5. 삭제 관련 manifest 확인**

spark.sql("SELECT \* FROM ...manifests").show()

* Iceberg는 manifest 파일에 어떤 레코드가 추가(insert)되었고,  
  어떤 레코드가 \*\*제거(delete)\*\*되었는지를 기록한다
* 여기서 deleted\_data\_files, added\_data\_files 등의 열을 통해 변화된 파일 수 확인 가능

**🧠 정리**

| **기능** | **설명** |
| --- | --- |
| DELETE FROM ... WHERE ... | 조건에 맞는 데이터 삭제 (Soft Delete 방식) |
| snapshots 조회 | 삭제가 snapshot으로 저장되었는지 확인 |
| manifests 조회 | 어떤 파일이 삭제 또는 수정되었는지 확인 |

**✅ Iceberg의 삭제 방식 특징**

* Iceberg는 데이터를 바로 물리적으로 지우지 않고, **새로운 snapshot을 만들어 삭제를 추적**한다
* 향후 시간여행(time travel) 기능을 통해 삭제 이전 상태로 되돌릴 수 있다

이 소스는 Iceberg에서 **DML 기반 삭제**와 **스냅샷 기반 변경 이력 관리**가 어떻게 작동하는지를 확인하는 예제이다.

**iceberg\_time\_travel.py 소스 분석**

이 PySpark 코드는 \*\*Apache Iceberg의 시간여행(time travel)\*\*과 **스냅샷 기반 롤백 기능**을 활용하여 테이블 상태를 과거로 되돌렸다가, 다시 최신 상태로 복원하는 과정을 실행한다.

**✅ 핵심 기능 요약**

**1. 스냅샷 커밋 시간 목록 조회**

snapshotTimes = spark.sql("SELECT committed\_at AS commitTime FROM ...snapshots ORDER BY commitTime").collect()

* Iceberg 테이블의 모든 스냅샷 커밋 시간을 수집
* 이후 시간여행용 타임스탬프 선택을 위해 저장

**2. as-of-timestamp로 과거 시점 조회**

spark.read.option("as-of-timestamp", snapshotTimeMillis).format("iceberg").load(...).show()

* snapshotTimes[1] 값을 기준으로 과거 시점의 테이블 상태를 조회
* Iceberg의 **시간여행(시점 조회)** 기능 사용

**3. 스냅샷 ID 조회 및 특정 스냅샷 읽기**

spark.read.option("snapshot-id", snapshotID).format("iceberg").load(...).show()

* snapshot-id를 직접 지정하여 해당 시점의 데이터 확인
* ID 기반 조회는 timestamp보다 **정밀한 제어**가 가능

**4. 현재 snapshot 히스토리 조회**

spark.sql("SELECT \* FROM ...history").show()

* 테이블에 적용된 모든 스냅샷의 **타임라인 기록** 조회
* 각 스냅샷의 생성 시각, snapshot ID, 작업 유형 등을 확인

**5. 스냅샷 롤백 (과거로 되돌리기)**

spark.sql("CALL ...system.rollback\_to\_snapshot('trips', snapshotID)")

* Iceberg 테이블을 **초기 snapshot ID 상태로 되돌림**
* 이후 SELECT로 테이블 내용을 확인하면 **예전 데이터로 복원**되어 있음

**6. 최신 snapshot으로 복원**

spark.sql("CALL ...system.set\_current\_snapshot('trips', latest\_snapshotID)")

* 최신 snapshot ID를 다시 "현재 상태"로 설정
* 테이블이 최신 데이터로 되돌아옴

**🧠 전체 기능 정리**

| **기능** | **설명** |
| --- | --- |
| as-of-timestamp | 타임스탬프 기준 시간여행 쿼리 |
| snapshot-id | 특정 스냅샷 ID로 데이터 조회 |
| history 테이블 | 모든 스냅샷의 커밋 로그 확인 |
| rollback\_to\_snapshot() | 테이블을 과거 특정 시점으로 되돌림 |
| set\_current\_snapshot() | 테이블을 최신 상태로 다시 복구 |

**✅ Iceberg 시간여행/스냅샷 관리 특징**

* 데이터 삭제/업데이트 이후에도 **모든 변경 이력은 snapshot으로 보관**
* 시간여행은 **데이터 정합성 검증, 오류 복구, 감사**에 매우 유용
* Iceberg는 내부적으로 **immutable data file + metadata tree** 구조를 활용하여 롤백이 빠르고 안정적임

이 코드는 Iceberg의 **버전 관리, 데이터 복원, 감사 추적** 기능을 종합적으로 실습할 수 있는 예제로 적절하다.

**log 분석 : Spark UI -> Executors -> driver 🡪 stdout클릭**

|  |
| --- |
| snapshotTimes:  Row(commitTime=datetime.datetime(2025, 4, 6, 9, 12, 23, 489000))  0  Row(commitTime=datetime.datetime(2025, 4, 6, 9, 12, 29, 265000))  1  Row(commitTime=datetime.datetime(2025, 4, 6, 10, 3, 6, 948000))  2  Row(commitTime=datetime.datetime(2025, 4, 6, 10, 11, 25, 735000))  3  Row(commitTime=datetime.datetime(2025, 4, 6, 10, 40, 4, 698000))  4  root  |-- trip\_id: integer (nullable = true)  |-- tstamp: timestamp (nullable = true)  |-- route\_id: string (nullable = true)  |-- destination: string (nullable = true)  |-- origin: string (nullable = true)  State of the table as of time: 2025-04-06 09:12:29.265000  +-------+-------------------+--------+-----------+  |trip\_id|tstamp |route\_id|destination|  +-------+-------------------+--------+-----------+  |1540960|2025-04-06 09:12:24|A |Seattle |  |1540961|2025-04-06 09:12:24|B |New York |  |1540962|2025-04-06 09:12:24|C |New Jersey |  |1540963|2025-04-06 09:12:24|D |Los Angeles|  |1540964|2025-04-06 09:12:24|E |Las Vagas |  +-------+-------------------+--------+-----------+  only showing top 5 rows  snapshots:  Row(snapshot=8330148965376747891)  Row(snapshot=5256310174651890780)  Row(snapshot=9184610389125467227)  Row(snapshot=9178447876033454357)  Row(snapshot=6499578508932893850)  +-------+-------------------+--------+-------------+  |trip\_id|tstamp |route\_id|destination |  +-------+-------------------+--------+-------------+  |0 |2025-04-06 09:12:05|A |Seattle |  |1 |2025-04-06 09:12:05|B |New York |  |2 |2025-04-06 09:12:05|C |New Jersey |  ….  |23 |2025-04-06 09:12:05|D |Los Angeles |  |24 |2025-04-06 09:12:05|E |Las Vagas |  +-------+-------------------+--------+-------------+  only showing top 25 rows  View history...  +--------------------+-------------------+-------------------+-------------------+  | made\_current\_at| snapshot\_id| parent\_id|is\_current\_ancestor|  +--------------------+-------------------+-------------------+-------------------+  |2025-04-06 09:12:...|8330148965376747891| NULL| true|  |2025-04-06 09:12:...|5256310174651890780|8330148965376747891| true|  |2025-04-06 10:03:...|9184610389125467227|5256310174651890780| true|  |2025-04-06 10:11:...|9178447876033454357|9184610389125467227| true|  |2025-04-06 10:40:...|6499578508932893850|9178447876033454357| true|  +--------------------+-------------------+-------------------+-------------------+  View current data state:  +-------+-------------------+--------+-------------+------------+  |trip\_id|tstamp |route\_id|destination |origin |  +-------+-------------------+--------+-------------+------------+  |3 |2025-04-06 10:10:51|D |Los Angeles |Portland |  |10 |2025-04-06 10:10:51|A |Seattle |Chicago |  ….  |501 |2025-04-06 10:10:51|B |New York |Minneapolis |  |521 |2025-04-06 10:10:51|B |New York |Minneapolis |  +-------+-------------------+--------+-------------+------------+  only showing top 20 rows  Revert to the very beginning state at snapshot id: 8330148965376747891  View data set reverted back to the first state...  +-------+-------------------+--------+-------------+------+  |trip\_id|tstamp |route\_id|destination |origin|  +-------+-------------------+--------+-------------+------+  |0 |2025-04-06 09:12:05|A |Seattle |NULL |  |1 |2025-04-06 09:12:05|B |New York |NULL |  …  |28 |2025-04-06 09:12:05|I |Miami |NULL |  |29 |2025-04-06 09:12:05|J |San Francisco|NULL |  +-------+-------------------+--------+-------------+------+  only showing top 30 rows  Revert to latest...  +-------+-------------------+--------+-------------+------------+  |trip\_id|tstamp |route\_id|destination |origin |  +-------+-------------------+--------+-------------+------------+  |3 |2025-04-06 10:10:51|D |Los Angeles |Portland |  |10 |2025-04-06 10:10:51|A |Seattle |Chicago |  ….  |213 |2025-04-06 10:10:51|D |Los Angeles |Portland |  |220 |2025-04-06 10:10:51|A |Seattle |Chicago |  +-------+-------------------+--------+-------------+------------+  only showing top 10 rows  View transaction history...  +-----------------------+-------------------+-------------------+-------------------+  |made\_current\_at |snapshot\_id |parent\_id |is\_current\_ancestor|  +-----------------------+-------------------+-------------------+-------------------+  |2025-04-06 09:12:23.489|8330148965376747891|NULL |true |  |2025-04-06 09:12:29.265|5256310174651890780|8330148965376747891|false |  |2025-04-06 10:03:06.948|9184610389125467227|5256310174651890780|false |  |2025-04-06 10:11:25.735|9178447876033454357|9184610389125467227|false |  |2025-04-06 10:40:04.698|6499578508932893850|9178447876033454357|false |  |2025-04-06 10:51:46.929|8330148965376747891|NULL |true |  +-----------------------+-------------------+-------------------+-------------------+ |

이 로그는 **Apache Iceberg 테이블의 스냅샷 기반 시간여행 및 롤백 흐름 전체**를 잘 보여준다.  
실제로 데이터를 삽입하고, 여러 번 수정하거나 삭제한 후 특정 시점으로 되돌리고 다시 최신 상태로 복구하는 **버전 관리 흐름을 시각화**한 것이다.

**✅ 전체 흐름 요약**

**1. ✅ 최초 스냅샷 생성 (snapshot 0)**

2025-04-06 09:12:23.489 → snapshot\_id: 8330148965376747891

* 테이블에 최초로 데이터를 insert한 시점이다.
* origin 컬럼은 아직 없어서 모두 NULL로 채워져 있음.
* 이 스냅샷은 기준 시점이므로 **parent\_id = NULL**, **is\_current\_ancestor = true**

**2. ✅ 두 번째 스냅샷 (snapshot 1)**

2025-04-06 09:12:29.265 → snapshot\_id: 5256310174651890780

* 이 시점에 일부 **추가 insert** 또는 **update**가 발생한 것으로 보인다.
* 데이터 수가 증가했고, trip\_id 값이 증가한 기록이 있음.
* origin 컬럼은 여전히 없음 또는 NULL 상태로 보인다.

**3. ✅ 이후 여러 변경 (snapshot 2~4)**

snapshot\_id: 9184610389125467227 (10:03:06)

snapshot\_id: 9178447876033454357 (10:11:25)

snapshot\_id: 6499578508932893850 (10:40:04)

* 이들 스냅샷에서는 MERGE INTO를 통한 UPSERT, DELETE, ALTER TABLE ADD COLUMN origin, UPDATE origin 등이 수행되었을 가능성이 높다.
* 이 시점 이후부터 origin 컬럼이 추가되고 값이 채워짐:

|trip\_id|destination |origin |

|-------|-------------|------------|

|3 |Los Angeles |Portland |

|10 |Seattle |Chicago |

**4. 🔁 롤백 발생 (snapshot 5)**

2025-04-06 10:51:46.929 → snapshot\_id: 8330148965376747891 (최초 snapshot)

* CALL rollback\_to\_snapshot(...) 명령으로 최초 상태로 되돌림
* 이 스냅샷은 기존에 존재하던 snapshot이지만 **다시 현재로 지정**되었기 때문에 히스토리에 다시 기록됨
* parent\_id = NULL, is\_current\_ancestor = true

**5. 🔄 최신 상태로 복구**

* CALL set\_current\_snapshot(...) 명령으로 최신 snapshot으로 다시 되돌림
* 복구 후 데이터 확인 결과, origin 컬럼이 포함된 최신 데이터가 다시 등장함

**✅ 핵심 구조 해석**

**📊 히스토리 테이블 (history)**

| **made\_current\_at** | **snapshot\_id** | **parent\_id** | **is\_current\_ancestor** |
| --- | --- | --- | --- |
| 09:12:23 | 8330...7891 (최초) | NULL | true |
| 09:12:29 | 5256...0780 | 8330...7891 | false |
| 10:03:06 | 9184...7227 | 5256...0780 | false |
| 10:11:25 | 9178...4357 | 9184...7227 | false |
| 10:40:04 | 6499...3850 | 9178...4357 | false |
| 10:51:46 | 8330...7891 | NULL | true |

* is\_current\_ancestor = true로 표시된 마지막 두 스냅샷은 **롤백 전의 최초 상태**와 **롤백 후 현재 상태**이다.
* Iceberg는 롤백할 때 **기존 snapshot을 재사용**하며, 이력이 겹치지 않도록 새로운 made\_current\_at 타임스탬프로 관리한다.

**✅ 결론 요약**

| **시점** | **동작** | **설명** |
| --- | --- | --- |
| 09:12:23 | 최초 insert | snapshot 0 생성 |
| 09:12:29 ~ 10:40:04 | insert, update, delete | snapshot 1~4 누적 |
| 10:51:46 | rollback | snapshot 0으로 되돌림 |
| 이후 | 최신 snapshot으로 복구 | 최신 데이터 재적용 |

이 로그는 Iceberg가 **스냅샷 기반으로 변경 이력을 안전하게 추적하고, 언제든 복원 가능한 구조**임을 잘 보여준다.

**iceberg\_partition\_evolution.py 소스 분석**

이 PySpark 코드는 **Iceberg 테이블에 시간 단위 파티션 필드를 추가하고**,  
**새로운 데이터를 생성해 해당 파티션 필드를 포함한 형태로 삽입하는 과정**을 포함한다.

**✅ 핵심 기능 요약**

**1. 기존 Iceberg 테이블 스키마 및 파티션 확인**

spark.sql("DESC ...").show()

spark.sql("SELECT \* FROM ...partitions").show()

* 현재 Iceberg 테이블에 설정된 **파티션 필드 확인**
* 기본적으로 days(tstamp)와 같은 날짜 단위 파티션만 존재했을 가능성이 높음

**2. 파티션 필드 추가: 시간 단위(hours(tstamp))**

spark.sql("ALTER TABLE ... ADD PARTITION FIELD hours(tstamp)")

* Iceberg는 **다중 파티션 필드**를 지원함
* 기존에 days(tstamp)가 있더라도, hours(tstamp)를 추가로 지정 가능
* 추가 후 다시 DESC, partitions 테이블을 확인하여 파티션 변화 확인

**3. 추가 데이터 생성 전 마지막 trip\_id 찾기**

df\_current.select(expr("max(trip\_id)")).show()

* 테이블에 있는 현재 최대 trip\_id 값을 확인하여, 그 다음 값부터 새로운 데이터를 생성
* get\_json\_data\_update(start, count, ...) 함수에 사용됨

**4. 추가 데이터 생성 및 삽입**

df = Producer.create\_json\_df\_update(...)

df.createOrReplaceTempView("tripTempView")

spark.sql("""

INSERT INTO ... SELECT trip\_id, to\_timestamp(tstamp,...) AS timestamp, ...

FROM tripTempView

""")

* 100건의 데이터를 생성하여 새로 추가
* 새로 생성된 데이터는 **시간 단위 파티션 필드에 맞춰 자동 분배됨**

**5. 삽입 결과 확인**

spark.sql("SELECT \* FROM ... ORDER BY trip\_id DESC LIMIT 105").show()

* 새로 삽입된 100건 + 기존 데이터 중 일부를 확인하여 정상 반영 여부 검증

**🧠 정리: 이 코드에서 중요한 점**

| **기능** | **설명** |
| --- | --- |
| ADD PARTITION FIELD hours(tstamp) | 시간 단위 파티셔닝 추가 |
| max(trip\_id) → +1 | 데이터 생성 시 기존 값과 겹치지 않도록 자동 증가 |
| INSERT INTO | Iceberg 테이블에 추가 데이터 삽입 |
| DESC + partitions 조회 | 파티션 필드 변경 사항 확인 |

✅ 이 코드의 목적

1. Iceberg 테이블에 새로운 파티션 필드를 추가하는 방법 실습
2. 파티션 기준에 맞춰 새로운 데이터를 삽입하는 흐름 이해
3. 파티션 필드가 쿼리 성능 및 관리 구조에 미치는 영향 확인 준비

**일별 분할을 보여주는 화면 샷**: 실제 다른 날로 실습해야 두개로 보임

S3에 한 개의 날짜만 생성

텍스트, 스크린샷, 폰트, 번호이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

**✅ 두 개 이상의 파티션 폴더가 생성되려면?**

Iceberg 테이블에 **날짜가 다른 데이터를 삽입**해야 한다.  
즉, tstamp 컬럼에 들어가는 값이 **다른 날짜여야만** Iceberg가 S3에 새로운 "tstamp\_day=..." 디렉토리를 만든다.

**log 분석 : Spark UI -> Executors -> driver 🡪 stdout클릭**

|  |
| --- |
| View schema and partition data:  +--------------+------------+-------+  | col\_name| data\_type|comment|  +--------------+------------+-------+  | trip\_id| int| NULL|  | tstamp| timestamp| NULL|  | route\_id| string| NULL|  | destination| string| NULL|  | origin| string| NULL|  | | | |  |# Partitioning| | |  | Part 0|days(tstamp)| |  +--------------+------------+-------+  +------------+-------+------------+----------+-----------------------------+----------------------------+--------------------------+----------------------------+--------------------------+--------------------+------------------------+  | partition|spec\_id|record\_count|file\_count|total\_data\_file\_size\_in\_bytes|position\_delete\_record\_count|position\_delete\_file\_count|equality\_delete\_record\_count|equality\_delete\_file\_count| last\_updated\_at|last\_updated\_snapshot\_id|  +------------+-------+------------+----------+-----------------------------+----------------------------+--------------------------+----------------------------+--------------------------+--------------------+------------------------+  |{2025-04-06}| 0| 1500000| 1| 4040717| 0| 0| 0| 0|2025-04-06 09:12:...| 8330148965376747891|  +------------+-------+------------+----------+-----------------------------+----------------------------+--------------------------+----------------------------+--------------------------+--------------------+------------------------+  Add partition field to include timestamp hour:  +--------------+-------------+-------+  | col\_name| data\_type|comment|  +--------------+-------------+-------+  | trip\_id| int| NULL|  | tstamp| timestamp| NULL|  | route\_id| string| NULL|  | destination| string| NULL|  | origin| string| NULL|  | | | |  |# Partitioning| | |  | Part 0| days(tstamp)| |  | Part 1|hours(tstamp)| |  +--------------+-------------+-------+  +------------+-------+------------+----------+-----------------------------+----------------------------+--------------------------+----------------------------+--------------------------+--------------------+------------------------+  | partition|spec\_id|record\_count|file\_count|total\_data\_file\_size\_in\_bytes|position\_delete\_record\_count|position\_delete\_file\_count|equality\_delete\_record\_count|equality\_delete\_file\_count| last\_updated\_at|last\_updated\_snapshot\_id|  +------------+-------+------------+----------+-----------------------------+----------------------------+--------------------------+----------------------------+--------------------------+--------------------+------------------------+  |{2025-04-06}| 0| 1500000| 1| 4040717| 0| 0| 0| 0|2025-04-06 09:12:...| 8330148965376747891|  +------------+-------+------------+----------+-----------------------------+----------------------------+--------------------------+----------------------------+--------------------------+--------------------+------------------------+  Insert additional rows of data...  Get last trip\_id value...  +------------+  |max(trip\_id)|  +------------+  | 1499999|  +------------+  <class 'int'>  Next trip\_id is: 1500000  Generate a dataset of {} rows  +-------------+------------+--------+-------+-------------------+  |destination |origin |route\_id|trip\_id|tstamp |  +-------------+------------+--------+-------+-------------------+  |Seattle |Chicago |A |1500000|2025-04-06 11:05:59|  |New York |Minneapolis |B |1500001|2025-04-06 11:05:59|  |New Jersey |Boston |C |1500002|2025-04-06 11:05:59|  …  |Philadelphia |Des Moines |H |1500017|2025-04-06 11:05:59|  |Miami |Kansas City |I |1500018|2025-04-06 11:05:59|  |San Francisco|Indianapolis|J |1500019|2025-04-06 11:05:59|  +-------------+------------+--------+-------+-------------------+  only showing top 20 rows  Insert the additional rows of data...  Show last 105 rows of data  +-------+-------------------+--------+-------------+------------+  |trip\_id|tstamp |route\_id|destination |origin |  +-------+-------------------+--------+-------------+------------+  |1500099|2025-04-06 11:05:59|J |San Francisco|Indianapolis|  |1500098|2025-04-06 11:05:59|I |Miami |Kansas City |  |1500097|2025-04-06 11:05:59|H |Philadelphia |Des Moines |  ….  |1499998|2025-04-06 09:12:05|I |Miami |NULL |  |1499997|2025-04-06 09:12:05|H |Philadelphia |NULL |  |1499996|2025-04-06 09:12:05|G |Washinton DC |NULL |  |1499995|2025-04-06 09:12:05|F |Tucson |NULL |  +-------+-------------------+--------+-------------+------------+ |

이 로그는 Iceberg 테이블에 대해 **시간(hour) 단위 파티션을 추가하고**,  
그에 맞게 데이터를 생성·삽입하여 파티션 동작을 확인한 전체 흐름을 잘 보여준다.

**✅ 1. 테이블 초기 상태 확인**

View schema and partition data:

|# Partitioning| Part 0|days(tstamp)|

* Iceberg 테이블은 처음에 days(tstamp)만 파티션 필드로 설정되어 있었다.
* 즉, 날짜(예: 2025-04-06) 단위로만 데이터가 분할되어 저장되었음

|{2025-04-06}| spec\_id=0 | record\_count=1500000 | ...

* 실제로 2025-04-06 하루에 해당하는 파티션 하나만 존재함

**✅ 2. hours(tstamp) 파티션 필드 추가**

Add partition field to include timestamp hour:

|# Partitioning| Part 0 | days(tstamp) |

| | Part 1 | hours(tstamp)|

* ALTER TABLE ... ADD PARTITION FIELD hours(tstamp) 실행됨
* 시간 단위로 추가 파티셔닝 가능해졌음 (예: 2025-04-06-11)

주의: 기존에 있던 데이터는 여전히 spec\_id=0 (days 단위 파티션 스펙)을 사용함

**✅ 3. 새 데이터 생성 및 삽입 준비**

Get last trip\_id value...

|max(trip\_id)| → 1499999 → Next trip\_id: 1500000

* 기존 데이터 중 가장 마지막 trip\_id를 확인하고, 충돌 없이 새 데이터를 만들기 위한 기준으로 사용함

**✅ 4. 새로운 데이터 100건 생성**

Generate a dataset of 100 rows

|trip\_id|tstamp |destination |origin |

|1500000|2025-04-06 11:05:59|Seattle |Chicago |

...

* tstamp 값이 **모두 2025-04-06 11:05:59**로 동일 → 파티션 필드에 따라 "2025-04-06-11"로 분류될 예정
* origin 컬럼도 잘 채워짐 (Des Moines, Indianapolis 등)

**✅ 5. INSERT INTO로 Iceberg 테이블에 삽입**

Insert the additional rows of data...

* 데이터 삽입 시 Iceberg가 새로운 partition spec(days, hours)에 맞춰 내부적으로 적절히 처리함
* tstamp를 기준으로 hours(tstamp) 파티션이 활성화됨

**✅ 6. 삽입 결과 확인**

Show last 105 rows of data

|trip\_id|tstamp |origin |

|1500099|2025-04-06 11:05:59|Indianapolis|

...

|1500000|2025-04-06 11:05:59|Chicago |

|1499999|2025-04-06 09:12:05|NULL |

* trip\_id=1500000부터 시작하는 행들은 모두 origin 값이 있고, **tstamp는 11:05:59**로 동일
* trip\_id=1499999 이하 데이터는 이전 snapshot에서 삽입된 값이며, origin은 아직 없음 (NULL)
* **삽입된 100건이 새 파티션(h=11)에 들어간 것**을 간접적으로 보여줌

**✅ 결론 요약**

| **항목** | **설명** |
| --- | --- |
| 기존 파티션 필드 | days(tstamp)만 존재 |
| 추가된 파티션 필드 | hours(tstamp) 추가됨 |
| 기존 데이터 | spec\_id=0, 하루 단위 파티션 |
| 새로 삽입된 데이터 | tstamp=2025-04-06 11:05:59, hours 단위 파티션 대상 |
| 효과 | Iceberg 테이블은 **기존 스냅샷과 새 스냅샷에 대해 서로 다른 파티션 스펙을 공존**시킴 |

**🧠 Iceberg 특징 정리**

* Iceberg는 **멀티 파티션 스펙**을 지원한다 (spec\_id로 식별)
* 파티션 필드를 추가하면 **기존 데이터는 그대로 유지**되고, **새 데이터부터 새 파티션 방식 적용**
* 이는 운영 중인 테이블에도 **유연하게 파티션 구조를 진화**시킬 수 있다는 것을 의미한다

**✅ Iceberg 테이블의 파티션 진화: 구조, 필요성, 활용**

**1. ✅ 파티션이란?**

* **대량의 데이터를 빠르게 필터링하기 위한 물리적 분할 단위**이다.
* 일반적인 테이블은 데이터를 전부 스캔해야 하지만, 파티션을 사용하면 **필요한 부분만 읽을 수 있다.**

**📌 예시:**

-- 파티션이 없는 경우

SELECT \* FROM trips WHERE tstamp BETWEEN '2023-03-01' AND '2023-03-02';

→ 테이블 전체 스캔

-- 파티션이 있는 경우

→ tstamp\_day=2023-03-01 디렉토리만 스캔

**2. ✅ Iceberg의 파티션 구조**

Iceberg는 기존 Hive 방식과 달리, **파티션 정보를 메타데이터 파일에 따로 저장**하며  
**파티션 컬럼을 테이블의 물리적 스키마와 분리**하여 관리한다.

**Iceberg 파티션 특징:**

| **기능** | **설명** |
| --- | --- |
| **자동 인식** | S3 디렉토리 경로를 강제하지 않음. 내부 메타데이터에 따라 처리 |
| **함수 기반 표현** | days(tstamp), hours(tstamp) 같은 파티셔닝 가능 |
| **스냅샷별 스펙 추적** | 데이터 삽입 시점마다 사용된 파티션 방식(spec\_id)을 추적함 |
| **파티션 진화 지원** | 운영 중에도 파티션 필드를 추가/변경 가능 |

**3. ✅ 파티션 진화란?**

**테이블을 운영 중에도 파티션 구조를 변경할 수 있는 기능**이다.  
예전 시스템(Hive, Hudi 등)에서는 파티션 스키마 변경 시 테이블을 재생성해야 했지만,  
Iceberg는 **스냅샷 기반 구조** 덕분에 **기존 데이터는 유지한 채, 새 파티션 구조를 추가 적용**할 수 있다.

**예시 흐름**

**🔹 기존:**

PARTITIONED BY days(tstamp)

**🔹 파티션 진화 (추가):**

ALTER TABLE trips ADD PARTITION FIELD hours(tstamp)

→ 이후 삽입되는 데이터부터 hours(tstamp)까지 적용됨  
→ 기존 데이터는 여전히 days(tstamp) 기준 유지됨

**4. ✅ 왜 필요한가? (파티션 진화의 필요성)**

| **이유** | **설명** |
| --- | --- |
| **쿼리 성능 향상** | 더 정밀한 필터링이 필요한 경우, 파티션 필드를 시간(hour), 지역 등으로 확장 가능 |
| **운영 유연성 확보** | 테이블 재생성 없이 파티션 구조 변경 가능 → 무중단 운영 가능 |
| **데이터 볼륨 변화 대응** | 데이터 양이 많아졌을 때 파티션을 세분화하여 병렬 처리 효율 증가 |
| **모델 변경 대응** | 비즈니스 요건 변경 시 유연하게 파티션 필드 교체 가능 |

**5. ✅ 실제 활용 예시**

**🧪 단계별 시나리오**

1. **초기 테이블 생성**

CREATE TABLE trips (

trip\_id INT,

tstamp TIMESTAMP,

destination STRING

)

USING iceberg

PARTITIONED BY (days(tstamp))

1. **파티션 필드 추가**

ALTER TABLE trips ADD PARTITION FIELD bucket(destination, 4)

1. **추가 데이터 삽입 시 적용**

* 새로 삽입된 데이터는 destination을 기준으로 bucket 파티션 분산됨

1. **과거 데이터는 여전히 days(tstamp) 기준**

* Iceberg는 스냅샷별로 어떤 파티션 스펙이 쓰였는지를 모두 기록하여 혼합 상태도 지원

**6. ✅ 정리**

| **항목** | **Iceberg 파티션 진화 특징** |
| --- | --- |
| 지원 여부 | ✅ 운영 중에도 가능 |
| 기존 데이터 영향 | ❌ 없음 (기존 파티션 구조 유지) |
| 변경 적용 범위 | 이후 삽입되는 데이터부터 |
| 메타데이터 기록 | spec\_id 기준으로 스냅샷별 추적 |
| 주요 활용 | 성능 향상, 운영 유연성, 쿼리 최적화 |

**🧠 결론**

**Iceberg의 파티션 진화는 운영 중인 대규모 데이터 테이블의 구조를 유연하게 확장할 수 있게 해주는 핵심 기능이다.**  
이를 통해 테이블을 재생성하거나 중단할 필요 없이, **더 정밀한 쿼리 필터링, 성능 향상, 비즈니스 모델 변화 대응**이 가능해진다.

**델타레이크**

deltalake.zip 파일을 다운받아 압축 풀고 안에 있는 하위 폴더 세 개를

기존 S3버킷 emrserverless-workshop-891377038690 밑에 업로드한다

[실습 주의 사항]

Delta Lake 버전 호환성 문제로 job 실행 시 오류 발생함

텍스트, 스크린샷, 폰트이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

**-Application 생성**

https://catalog.us-east-1.prod.workshops.aws/workshops/e8e8fbb5-c3fb-4f86-bf77-0ba1fe402c55/en-US/submit-jobs/spark-job/spark-job-studio

**위 실습 맨처음 페이지의 Application 생성에서 spark Release 버전을 emr-6.7.0으로 해서**

**Application을 다시 만든다 (Name: my-serverless-application-emr670 )**

**텍스트, 스크린샷, 번호, 폰트이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.**

나머지는 그대로 두고 [Create and start application] 클릭

Job 설정 시 아래 내용 복사해서

Spark properties 에서 Edit in text에 한꺼번에 붙여 넣는다 (계정ID는 각자 수정)

|  |
| --- |
| --jars s3://emrserverless-workshop-891377038690/jars/delta-core\_2.12-2.0.0.jar,s3://emrserverless-workshop-891377038690/jars/delta-storage-2.0.0.jar --conf spark.sql.extensions=io.delta.sql.DeltaSparkSessionExtension --conf spark.sql.catalog.spark\_catalog=org.apache.spark.sql.delta.catalog.DeltaCatalog --conf spark.archives=s3://emrserverless-workshop-891377038690/dependencies/pyspark\_deltalake.tar.gz#environment --conf spark.emr-serverless.driverEnv.PYSPARK\_DRIVER\_PYTHON=./environment/bin/python --conf spark.emr-serverless.driverEnv.PYSPARK\_PYTHON=./environment/bin/python --conf spark.emr-serverless.executorEnv.PYSPARK\_PYTHON=./environment/bin/python |

(이유 : --conf spark.jars=s3://.../delta-core.jar,s3://.../delta-storage.jar 로 설정하면

일부 Spark 버전/플랫폼에서는 .jar을 실행하려고 시도 → No main class 오류)

나머지 설정은 실습 지시대로 설정하고 [Submit job run] 클릭

**✅ emr-7.8.0은 어떤 Spark 버전을 포함하나?**

AWS 공식 문서 기준으로:

**EMR 7.8.0** → **Apache Spark 3.4.1** 포함

**✅ 권장 Delta Lake 버전**

| **Spark 버전** | **Delta Lake 호환 버전** |
| --- | --- |
| 3.2.x | Delta 2.0.0~2.2.0 |
| 3.3.x | Delta 2.3.0~2.4.0 |
| 3.4.x | ✅ **Delta 2.4.0~3.0.0+** ← emr-7.8.0 사용시 |

아래와 같이 cloud9에서 직접 명령어로도 실행 가능하다(계정 ID는 변경)

|  |
| --- |
| aws emr-serverless start-job-run \  --application-id 00frhp5ttau5b509 \  --execution-role-arn arn:aws:iam::891377038690:role/EMRServerlessS3RuntimeRole \  --job-driver '{  "sparkSubmit": {  "entryPoint": "s3://emrserverless-workshop-891377038690/scripts/deltalake\_create.py",  "entryPointArguments": ["emrserverless-workshop-891377038690"],  "sparkSubmitParameters": "--jars s3://emrserverless-workshop-891377038690/jars/delta-core\_2.12-2.0.0.jar,s3://emrserverless-workshop-891377038690/jars/delta-storage-2.0.0.jar --conf spark.sql.extensions=io.delta.sql.DeltaSparkSessionExtension --conf spark.sql.catalog.spark\_catalog=org.apache.spark.sql.delta.catalog.DeltaCatalog --conf spark.archives=s3://emrserverless-workshop-891377038690/dependencies/pyspark\_deltalake.tar.gz#environment --conf spark.emr-serverless.driverEnv.PYSPARK\_DRIVER\_PYTHON=./environment/bin/python --conf spark.emr-serverless.driverEnv.PYSPARK\_PYTHON=./environment/bin/python --conf spark.emr-serverless.executorEnv.PYSPARK\_PYTHON=./environment/bin/python"  }  }' |

**deltalake\_create.py 소스 분석**

**S3에 Delta Lake 형식으로 데이터를 저장하고, Athena에서 조회할 수 있도록 manifest 생성까지 수행하는 예제이다.**

**🔧 단계별 요약 분석**

**1. 입력 인자 처리**

**s3\_bucket = sys.argv[1]**

* **S3 버킷 이름을 인자로 받아서 Delta 테이블 저장 경로 구성에 사용**

**2. SparkSession + Delta 구성**

**builder = SparkSession.builder.appName(...).enableHiveSupport()**

**spark = configure\_spark\_with\_delta\_pip(builder).getOrCreate()**

* **Delta를 사용하는 Spark 세션 구성**
* **delta-spark 패키지가 포함되어 있어야 정상 실행됨**

**3. 데이터 생성**

**get\_json\_data(start, count, dest)**

* **목적지 리스트 dest를 기반으로 JSON 데이터 200만 건 생성**
* **각 레코드에는 trip\_id, tstamp, route\_id, destination 포함**

**4. Spark DataFrame 변환**

**create\_json\_df(spark, data)**

* **JSON 데이터를 RDD로 변환 후 DataFrame으로 로딩**

**5. Delta 형식으로 저장**

**df1.write.mode("overwrite").format("delta").partitionBy("route\_id").save(deltaHivePath)**

* **지정된 S3 경로에 Delta 형식으로 저장**
* **route\_id 기준으로 파티셔닝**

**6. Spark SQL로 검증**

**spark.read.format("delta").load(...).createOrReplaceTempView("temp\_trip\_table")**

* **Delta 테이블을 임시 뷰로 등록한 뒤 SQL 질의 수행 (count, limit, max)**

**7. Athena를 위한 manifest 파일 생성**

**deltaTable.generate("symlink\_format\_manifest")**

* **Athena에서 Delta 테이블을 Parquet External Table로 조회 가능하도록 symlink 생성**

**📌 최종 산출물**

| **산출물** | **위치 / 형태** |
| --- | --- |
| **Delta 테이블** | **s3://{bucket}/delta-hive-table/** |
| **symlink manifest** | **.../delta-hive-table/\_symlink\_format\_manifest/** |
| **Athena 연결용 테이블** | **CREATE EXTERNAL TABLE ... 가능 (별도 SQL)** |

**⚠️ 전제 조건**

* **.tar.gz 환경에 delta-spark 포함되어 있어야 함**
* **Delta 관련 JAR이 Spark에 등록되어 있어야 함 (via --jars)**
* **Glue Data Catalog는 사용하지 않음 (Athena 직접 테이블 생성 방식)**

**✅ 요약**

**이 스크립트는 200만 개의 JSON 데이터를 Delta Lake 테이블로 저장하고, Athena에서 읽을 수 있도록 symlink manifest를 생성하는 전형적인 EMR Serverless + Delta 실습용 예제이다.**

**Table 생성 소스**

|  |
| --- |
| CREATE EXTERNAL TABLE my\_delta\_db.delta\_trip\_table (  destination string,  trip\_id bigint,  tstamp string,  origin string  )  PARTITIONED BY (route\_id string)  ROW FORMAT SERDE 'org.apache.hadoop.hive.ql.io.parquet.serde.ParquetHiveSerDe'  STORED AS INPUTFORMAT 'org.apache.hadoop.hive.ql.io.SymlinkTextInputFormat'  OUTPUTFORMAT 'org.apache.hadoop.hive.ql.io.HiveIgnoreKeyTextOutputFormat'  **LOCATION 's3:// emrserverless-workshop-891377038690/delta-hive-table/\_symlink\_format\_manifest/';** |

**🔍 전체 목적**

Delta Lake 테이블에서 generate("symlink\_format\_manifest")로 생성된 Symlink 기반 Parquet 파일 목록을 Athena 외부 테이블로 등록하는 SQL이다.

🧩 구문 분석

CREATE EXTERNAL TABLE my\_delta\_db.delta\_trip\_table (

destination string,

trip\_id bigint,

tstamp string,

origin string

)

* 외부 테이블 이름: my\_delta\_db.delta\_trip\_table
* 컬럼 정의: Delta 테이블에 저장된 각 JSON 레코드의 필드

PARTITIONED BY (route\_id string)

* Delta 테이블이 route\_id 기준으로 파티셔닝 되어 저장되었기 때문에 반드시 지정
* 파티션 열 누락 시 → 쿼리 오류 혹은 빈 결과 발생

ROW FORMAT SERDE 'org.apache.hadoop.hive.ql.io.parquet.serde.ParquetHiveSerDe'

* Athena가 Parquet 포맷을 이해하도록 지정
* Delta는 내부적으로 Parquet이므로 이 SerDe를 사용해야 함

STORED AS INPUTFORMAT 'org.apache.hadoop.hive.ql.io.SymlinkTextInputFormat'

OUTPUTFORMAT 'org.apache.hadoop.hive.ql.io.HiveIgnoreKeyTextOutputFormat'

* 핵심 설정:
  + SymlinkTextInputFormat은 \_symlink\_format\_manifest/ 안의 symlink 파일 목록을 파싱해서 Parquet 파일 경로를 가져옴
  + HiveIgnoreKeyTextOutputFormat은 출력 포맷으로 거의 항상 사용됨 (읽기 전용이므로 중요하지 않음)

LOCATION 's3://emrserverless-workshop-891377038690/delta-hive-table/\_symlink\_format\_manifest/';

* generate("symlink\_format\_manifest") 호출 후 생성된 경로
* 이 경로 안에는 \_symlink.txt 같은 symlink 파일이 존재하며, 그 안에 실제 데이터 파일 경로가 있음

✅ 요약

| 항목 | 설명 |
| --- | --- |
| 이 SQL의 목적 | Delta 테이블을 Athena에서 Parquet 테이블처럼 조회하기 위함 |
| 전제 조건 | Delta 테이블 생성 + symlink manifest 생성 완료 |

**MSCK REPAIR TABLE my\_delta\_db.delta\_trip\_table;** 명령은:

**S3에 존재하는 파티션 디렉터리들을 Glue Data Catalog 테이블의 메타데이터에 자동으로 추가**하는 명령이다.

**✅ MSCK 의미**

| **약어** | **뜻** |
| --- | --- |
| **MSCK** | **Metastore Check** |

**🧾 역할**

MSCK REPAIR TABLE은 Hive 또는 AWS Glue의 **메타스토어(Metastore)** 정보를 실제 S3 폴더 구조와 비교해서,  
**누락된 파티션 정보를 자동으로 동기화(Repair)** 하는 명령이다.

즉, **메타스토어를 점검하고 보수(repair)** 하는 기능이라 Metastore Check라고 부른다.

**✅ 주요 기능 요약**

* PARTITIONED BY (...) 되어 있는 Hive 또는 Glue 테이블에서
* 테이블 생성 후 추가된 **파티션 폴더들을 자동으로 인식해서 등록**해준다
* 예: s3://.../table/route\_id=A/, route\_id=B/ 같은 구조

**📦 전제 조건**

* 테이블은 **Glue Catalog에 등록되어 있어야 함**
* 테이블은 **PARTITIONED BY** 컬럼을 갖고 있어야 함
* 파티션은 **S3 디렉터리 구조로 존재**해야 함

**❗ 주의: Symlink 기반 테이블에는 적용되지 않음**

→ Delta Lake + Athena Symlink 방식은 Glue에 파티션 등록을 하지 않기 때문

**Delta 테이블을 사용하여 데이터 삽입 및 스키마 진화 수행 실습**

SELECT count(\*) FROM my\_delta\_deb.delta\_trip\_table; (실습 내용 오타)

텍스트, 폰트, 스크린샷, 화이트이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

SELECT count(\*) FROM my\_delta\_db.delta\_trip\_table; 으로 수정해서 사용한다

**deltalake\_insert.py 소스 분석**

**✅ 핵심 목적**

기존 Delta Lake 테이블에 **origin 컬럼이 추가된 새로운 데이터(20개)를 insert**하고,  
**Athena용 symlink manifest를 재생성**한다.

**🔍 단계별 핵심 요약**

**1. 🔧 기존 Delta 테이블 경로 지정**

deltaHivePath = "s3://{bucket}/delta-hive-table/"

* 테이블은 이미 이 위치에 존재함 (overwrite 아님)

**2. 🧾 새로운 데이터 생성**

get\_json\_data\_update(...)

* **총 20건의** trip\_id, tstamp, route\_id, destination, origin 컬럼이 포함된 데이터 생성

**3. 🧱 DataFrame 생성**

tripUpdates = create\_json\_df\_update(...)

* JSON 데이터를 Spark DataFrame으로 변환

**4. ➕ Delta 테이블에 append**

tripUpdates.write.format("delta")

.mode("append")

.option("mergeSchema", "true")

.partitionBy("route\_id")

.save(deltaHivePath)

| **옵션** | **설명** |
| --- | --- |
| mode("append") | 기존 데이터 유지하면서 새 데이터 추가 |
| mergeSchema=true | 기존 테이블에 없던 컬럼(origin)도 추가 허용 |
| partitionBy("route\_id") | 파티션 기준은 여전히 동일함 |

**5. 🔁 Athena manifest 갱신**

deltaTable = DeltaTable.forPath(spark, deltaHivePath)

deltaTable.generate("symlink\_format\_manifest")

* 새로 추가된 파일을 포함하도록 \_symlink\_format\_manifest/ 디렉토리 재생성

이걸 하지 않으면 Athena는 **추가된 데이터를 조회하지 못함**

**✅ 한 문장 요약**

이 스크립트는 기존 Delta 테이블에 origin 컬럼이 포함된 데이터를 **append + 스키마 자동 병합**하고, Athena에서 이를 읽을 수 있도록 **manifest를 업데이트**하는 작업을 수행한다.

**Delta 테이블의 레코드 업데이트**

delta-lake-update 이름으로 하고 deltalake\_update.py 를 설정하여 job을 실행한다

**deltalake\_update.py 소스 분석**

이 스크립트는 앞의 Delta 테이블 실습 흐름 중 데이터 수정(update)에 해당하는 핵심 로직이다.

**✅ 핵심 목적**

Delta 테이블에서 destination = 'Syracuse' 인 데이터를 **Philadelphia로 일괄 수정**하고,  
수정된 데이터를 반영하여 **Athena용 symlink manifest를 재생성**한다.

**🔍 핵심 단계 요약**

**1. 🔎 현재 상태 조회**

spark.read.format("delta").load(...).createOrReplaceTempView("temp\_trip\_table")

spark.sql("SELECT count(\*) FROM temp\_trip\_table WHERE destination = 'Syracuse'")

* Delta 테이블에서 Syracuse, Philadelphia 건수 확인

**2. 🧱 DeltaTable 객체 가져오기**

deltaTable = DeltaTable.forPath(spark, deltaHivePath)

* Delta API를 사용하기 위한 객체 생성

**3. ✏️ 데이터 업데이트**

deltaTable.update("destination = 'Syracuse'", {"destination": "'Philadelphia'"})

| **항목** | **설명** |
| --- | --- |
| 조건 | destination = 'Syracuse' |
| 변경 | destination = 'Philadelphia' 으로 전체 치환 |
| 방식 | Delta Lake 내부적으로 **merge + copy-on-write** 방식으로 처리됨 |

**4. 🔁 Manifest 갱신**

deltaTable.generate("symlink\_format\_manifest")

* 업데이트된 Delta 파일을 기준으로 새로운 symlink 목록 생성
* Athena에서 SELECT 시 최신 내용 반영됨

**✅ 한 문장 요약**

이 스크립트는 Delta 테이블에서 **Syracuse를 Philadelphia로 일괄 변경**한 뒤,  
**Athena가 이를 인식하도록 symlink manifest를 재생성**하는 업데이트 처리 예제이다.

**💡 팁**

| **목적** | **명령** |
| --- | --- |
| 변경 후 확인 | SELECT destination, count(\*) FROM temp\_trip\_table GROUP BY destination |
| Athena에서 즉시 반영되도록 | generate("symlink\_format\_manifest")는 반드시 호출해야 함 |

**Delta 테이블의 레코드 삭제**

delta-lake-delete 이름으로 하고 deltalake\_delete.py 를 설정하여 job을 실행한다

deltalake\_delete.py 소스 분석

이 스크립트는 Delta Lake 테이블에서 **특정 조건의 데이터(200000개)를 삭제(delete)** 하고,  
Athena에서 최신 상태를 조회할 수 있도록 **manifest 파일을 재생성**하는 예제이다.

**✅ 핵심 목적**

Delta 테이블에서 destination = 'New Jersey'인 데이터를 **영구 삭제**하고,  
**Athena에서 이를 반영할 수 있도록 symlink manifest를 갱신**한다.

**🔍 단계별 핵심 요약**

**1. 📊 삭제 전 상태 확인**

spark.sql("SELECT count(\*) FROM temp\_trip\_table")

spark.sql("SELECT count(\*) FROM temp\_trip\_table WHERE destination = 'New Jersey'")

* 전체 레코드 수 및 'New Jersey' 건수 확인용

**2. 🧱 DeltaTable 객체 생성**

deltaTable = DeltaTable.forPath(spark, deltaHivePath)

* Delta API를 사용하기 위한 객체 생성

**3. ❌ 데이터 삭제**

deltaTable.delete("destination = 'New Jersey'")

| **항목** | **설명** |
| --- | --- |
| 삭제 조건 | destination = 'New Jersey' |
| 처리 방식 | Delta의 **ACID 트랜잭션 기반** 삭제 (파일 단위 copy-on-write) |
| 결과 | 해당 조건을 만족하는 모든 레코드 삭제됨 |

**4. 🔁 Athena용 manifest 재생성**

deltaTable.generate("symlink\_format\_manifest")

* 삭제된 데이터가 Athena 결과에 반영되도록 symlink 파일 재작성

**5. 📉 삭제 후 결과 검증**

spark.sql("SELECT count(\*) FROM temp\_trip\_table")

spark.sql("SELECT count(\*) FROM temp\_trip\_table WHERE destination = 'New Jersey'")

* 전체 건수 및 'New Jersey' 건수가 줄어든 것을 확인

**✅ 한 문장 요약**

이 스크립트는 Delta Lake 테이블에서 destination = 'New Jersey'인 데이터를 삭제하고,  
삭제 후 결과가 Athena에서 반영되도록 symlink manifest를 다시 생성한다.

**💡 참고**

| **항목** | **내용** |
| --- | --- |
| Delta의 delete | 내부적으로 새로운 파일 생성 + 삭제된 버전 추적 |
| manifest 재생성 | 삭제나 업데이트 후 반드시 필요 (Athena는 symlink 기준으로 조회하기 때문) |
| Athena에서 효과 | \_symlink\_format\_manifest 위치에 접근하는 외부 테이블이 최신 상태 반영함 |

**Delta 테이블에 업서트**

delta-lake- upsert 이름으로 하고 deltalake\_ upsert.py 를 설정하여 job을 실행한다

**deltalake\_upsert.py  소스 분석**

이번 스크립트는 Delta Lake의 **Upsert (Merge) 기능을 활용하여** 기존 데이터를 업데이트하거나 새로운 데이터를 삽입하는 예제이다.

**✅ 핵심 목적**

trip\_id = 2000015 ~ 2000024 범위의 데이터를 Delta Lake 테이블에 **업서트(Upsert)** 하고,  
**Athena가 이를 반영하도록 symlink manifest를 재생성**한다.

**🔍 핵심 단계 요약**

**1. 📊 기존 테이블 상태 확인**

spark.sql("SELECT max(trip\_id) FROM temp\_trip\_table")

spark.sql("SELECT \* FROM temp\_trip\_table WHERE trip\_id > 2000014")

* 업데이트/삽입 대상의 기존 상태를 확인
* 향후 upsert 결과 확인을 위한 기준점

**2. 🧾 새로운 데이터 생성**

get\_json\_data\_update(2000015, 10, insert\_dest, insert\_origin)

* trip\_id 2000015 ~ 2000024 (10건)
* origin은 모두 Seattle, destination은 동부 도시들 (Stamford, Fairfield 등)

**3. 🧱 DataFrame 생성**

tripUpdates1 = create\_json\_df\_update(...)

* JSON 데이터를 Spark DataFrame으로 변환

**4. 🔁 Delta 테이블에 Upsert (MERGE)**

deltaTable.merge(...).whenMatchedUpdateAll().whenNotMatchedInsertAll().execute()

| **동작** | **설명** |
| --- | --- |
| merge(..., 't.trip\_id = s.trip\_id') | trip\_id 기준으로 join |
| whenMatchedUpdateAll() | 기존 trip\_id 있으면 → 모든 필드 업데이트 |
| whenNotMatchedInsertAll() | 없으면 → 새로 삽입 |
| execute() | 실행 |

**5. 🔄 Manifest 재생성**

deltaTable.generate("symlink\_format\_manifest")

* 업데이트/삽입된 결과가 Athena에서도 조회되도록 symlink 목록 재작성

**6. 👀 결과 확인**

spark.sql("SELECT \* FROM temp\_trip\_table WHERE trip\_id > 2000010")

* 새로운 trip\_id 범위(2000015~2000024) 확인
* 기존 데이터가 업데이트 되었는지 시각적으로 검증
* trip\_id 2000015에서 2000019까지 업데이트, trip\_id 2000020에서 2000024까지 새로운 레코드가 추가

**✅ 한 문장 요약**

이 스크립트는 trip\_id를 기준으로 Delta Lake 테이블에 조건부 업데이트 또는 삽입(upsert)을 수행하고,  
**Athena가 이를 반영할 수 있도록 symlink manifest를 재생성**하는 예제이다.

**💡 참고 포인트**

| **항목** | **내용** |
| --- | --- |
| Upsert 조건 | trip\_id가 기존에 존재하면 업데이트, 없으면 삽입 |
| merge() 사용 | Delta Lake의 대표적인 기능 중 하나 (ACID 보장) |
| generate() 필수 | Athena 외부 테이블에서 최신 내용 조회하려면 항상 호출 필요 |

**델타 테이블로 시간 여행하기**

delta-lake-transaction-history이름으로 하고 deltalake\_transaction\_history.py 를 설정하여 job을 실행한다

**deltalake\_transaction\_history.py 소스 분석**

이번 스크립트는 \*\*Delta Lake의 타임라인 기능(트랜잭션 로그)을 조회하는 예제이다.

**✅ 핵심 목적**

Delta Lake 테이블의 변경 이력(트랜잭션 히스토리)을 조회하여

**🔍 단계별 핵심 요약**

**1. 📁 Delta 테이블 경로 설정**

deltaHivePath = "s3://{bucket}/delta-hive-table/"

* 대상 Delta 테이블 경로

**2. 📌 DeltaTable 객체 생성**

deltaTable = DeltaTable.forPath(spark, deltaHivePath)

* Delta의 트랜잭션 API를 사용하기 위한 테이블 핸들 생성

**3. 📜 트랜잭션 히스토리 조회**

deltaTable.history(100)

* 최근 100개의 트랜잭션 로그 조회
* 생략 시 history()는 기본 1개만 보여줌

**4. 📊 특정 컬럼만 출력**

.select("version", "timestamp", "operation", "operationParameters")

.show(truncate=False)

| **컬럼** | **설명** |
| --- | --- |
| version | 트랜잭션 버전 (자동 증가) |
| timestamp | 작업이 수행된 시각 |
| operation | 수행된 작업 종류 (WRITE, MERGE, UPDATE, DELETE 등) |
| operationParameters | 파티션 정보, overwrite 여부 등 세부 설정 정보 |

**✅ 한 문장 요약**

이 스크립트는 Delta 테이블의 버전별 작업 이력(타임라인)을 최대 100개까지 조회하여  
어떤 시점에 어떤 작업이 수행되었는지 파악하는 데 사용된다.

**💡 활용 예시**

| **목적** | **활용** |
| --- | --- |
| 실수로 덮어쓴 데이터 시점 확인 | 특정 version 기록을 보고 복구 가능 |
| 운영 중 Merge/Update/Delete 작업 이력 추적 | 작업자 없이도 자동 추적 가능 |
| Time Travel 용 버전 식별 | .option("versionAsOf", X) 로 과거 데이터 조회 시 사용 |

**출력 로그 확인하는 방법**

Spark UI -> Executors -> driver -> stdout 클릭

텍스트, 스크린샷, 폰트, 번호이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

deltalake\_time\_travel.py 소스를 아래와 같이 수정하고 s3버킷에 업로드한다

|  |
| --- |
| df\_v0 = (spark.read  .format("delta")  .option("timestampAsOf", "**2025-04-06 14:59:09**") # timestamp after table creation  .load(deltaHivePath))  df\_v0.take(5) |

delta-lake-time-travel 이름으로 하고 deltalake\_time\_travel.py 를 설정하여 job을 실행한다

**deltalake\_time\_travel.py 소스 분석**

이 스크립트는 Delta Lake의 **Time Travel 기능**을 실습하는 예제이다.  
이전 버전의 Delta 테이블을 **timestamp 또는 버전(version) 기준으로 조회**하는 방식이다.

**✅ 핵심 목적**

Delta 테이블의 변경 이력을 기반으로 **과거 버전의 데이터를 시점별로 조회**하고,  
각각의 시점에서 데이터가 어떻게 바뀌었는지 비교한다.

**🔍 단계별 핵심 요약**

**1. 📜 트랜잭션 히스토리 확인**

deltaTable.history(100).select(...).show()

* 어떤 시점(version), 어떤 작업(INSERT, UPDATE, DELETE 등)이 있었는지 확인

**2. 🕰️ timestamp 기반 Time Travel**

.option("timestampAsOf", "2025-04-06 14:59:09")

* 해당 시점 이후로 존재하던 테이블 상태를 로딩
* 실시간으로 변화하는 데이터를 **지정 시간 기준으로 되돌려서 조회**할 수 있음

**3. 🔢 버전 기반 Time Travel**

.option("versionAsOf", "1")

.option("versionAsOf", "2")

.option("versionAsOf", "3")

* Delta가 관리하는 내부 버전(\_delta\_log) 기준으로 조회
* version 1 → 테이블 초기 생성
* version 2 → 예: 데이터 upsert
* version 3 → 예: 특정 레코드 삭제 등

**4. 🔎 시점별 데이터 조회**

* 버전마다 임시 뷰 생성 후 SQL 조회 수행

예:

SELECT count(\*) as Count\_total FROM temp\_trip\_table\_v3

SELECT count(\*) as Count\_NJ FROM temp\_trip\_table\_v3 WHERE destination = 'New Jersey'

* 각 버전에서 데이터 수, 특정 조건 필터 결과를 비교

**🧩 Time Travel 시나리오 예시 흐름**

| **작업 시점** | **내 용** |
| --- | --- |
| Version 0 | 최초 데이터 insert (trip\_id 2000000 ~) |
| Version 1 | append or update (e.g. origin 추가) |
| Version 2 | update: 'Syracuse' → 'Philadelphia' |
| Version 3 | delete: destination = 'New Jersey' |

**✅ 요약**

이 스크립트는 Delta Lake의 **Time Travel 기능을 사용하여**,  
**버전별 또는 시점별로 테이블 상태를 복원/조회**하고, 데이터 변경 내역을 시각적으로 분석하는 실습이다.

**💡 추가 팁**

| **기능** | **방법** |
| --- | --- |
| 특정 컬럼 기준 변경 추적 | SELECT trip\_id, destination FROM ... ORDER BY trip\_id |
| 타임트래블 + 필터링 결합 | .option("versionAsOf", 2).load(...).filter("destination = 'Syracuse'") |
| 최신 버전 확인 | deltaTable.history().select("version").head(1) |

**테이블을 버전 0으로 되돌리기**

delta-lake-revert-version-0 이름으로 하고 deltalake\_revert\_to\_version\_0.py 를 설정하여 job을 실행한다

**deltalake\_revert\_to\_version\_0.py 소스 분석**

이 스크립트는 Delta Lake의 **버전 복구 기능**인 restoreToVersion()을 활용하여 테이블을 \*\*초기 상태(버전 0)\*\*로 되돌리는 예제이다.

**✅ 핵심 목적**

Delta 테이블을 버전 0(최초 상태)\*로 복원하고,  
복원된 데이터를 **Athena에서 즉시 조회할 수 있도록 symlink manifest를 재생성**한다.

**🔍 단계별 핵심 요약**

**1. 📍 DeltaTable 객체 연결**

deltaTable = DeltaTable.forPath(spark, deltaHivePath)

* 지정된 S3 Delta 테이블 경로에 연결

**2. ⏪ 버전 복원 실행**

deltaTable.restoreToVersion(0)

| **항목** | **설명** |
| --- | --- |
| restoreToVersion(0) | Delta 테이블을 버전 0(초기 상태)로 복원 |
| 작동 방식 | Delta는 이전 버전 메타정보를 기반으로 현재 상태를 overwrite함 |
| 결과 | 최신 버전이 새로 생성되며, 내용은 버전 0과 동일해짐 |

❗ **물리적으로 과거 파일을 복사해서 덮어씌움** (delete 아님)

**3. 🔄 Symlink manifest 재생성**

deltaTable.generate("symlink\_format\_manifest")

* 복원된 내용을 Athena에서 확인할 수 있도록 symlink 파일 갱신
* 이전 버전의 symlink 경로는 유지되지만, symlink 내부 내용이 새로운 버전으로 대체됨

**✅ 한 문장 요약**

이 스크립트는 Delta 테이블을 **버전 0의 상태로 되돌리고**,  
Athena에서 복원된 데이터를 바로 조회할 수 있도록 **symlink manifest를 갱신**한다.

**💡 참고 정보**

| **항목** | **설명** |
| --- | --- |
| 복원 가능한 버전 조회 | deltaTable.history().select("version") |
| 특정 시간으로 복원 | .restoreToTimestamp("YYYY-MM-DD HH:MM:SS") 가능 |
| 복원 후 버전 | 새로운 버전이 생기며, 내용은 과거 기준 |

**🔐 주의사항**

* restoreToVersion()은 되돌리는 게 아니라 되돌린 내용을 기반으로 새 버전 생성"임
* 타임트래블로만 조회했던 이전 상태를 **정식 데이터로 되살리는 작업**임

**테이블을 최신 버전으로 되돌리기**

delta-lake-revert-to-latest-version 이름으로 하고 deltalake\_revert\_to\_latest\_version.py를 설정하여 job을 실행한다

**deltalake\_revert\_to\_latest\_version.py. 소스 분석**

이 스크립트는 Delta Lake의 restoreToVersion(4) 명령을 통해 **지정 버전(4번)의 상태로 데이터를 복구**하고,그 복구된 내용을 Athena에서 볼 수 있도록 **symlink manifest 파일을 갱신**하는 예제이다.

**✅ 핵심 목적**

Delta 테이블을 버전 4의 상태로 복원한 뒤,  
Athena에서 복원된 데이터를 확인할 수 있도록 symlink manifest를 다시 생성한다.

**🔍 단계별 핵심 요약**

**1. 📌 DeltaTable 객체 생성**

deltaTable = DeltaTable.forPath(spark, deltaHivePath)

* Delta Lake 테이블에 연결 (지정된 S3 경로 기준)

**2. ⏪ Delta 테이블 복원**

deltaTable.restoreToVersion(4)

| **항목** | **설명** |
| --- | --- |
| 복원 대상 | 버전 4 (version = 4)의 데이터 상태 |
| 작동 방식 | 해당 버전의 snapshot을 현재 상태로 복사해서 새로운 버전 생성 |
| 결과 | 가장 최신 버전은 복원된 데이터 기준으로 생성됨 (예: version 6이 마지막이면 → version 7 생성됨) |

❗ 실수로 잘못된 수정/삭제가 이뤄졌을 때 되돌리는 데 유용함

**3. 🔄 symlink manifest 재생성**

deltaTable.generate("symlink\_format\_manifest")

| **기능** | **설명** |
| --- | --- |
| 목적 | 복원된 데이터가 Athena 외부 테이블에서도 반영되도록 함 |
| 동작 | \_symlink\_format\_manifest/ 경로에 새로운 symlink 목록 파일 생성 |

**✅ 한 문장 요약**

이 스크립트는 Delta 테이블을 **버전 4의 상태로 복원**하고,  
**Athena에서 이 복원된 데이터를 확인할 수 있도록 symlink 파일을 재생성**한다.

**📌 실무 팁**

| **항목** | **권장 방식** |
| --- | --- |
| 버전 번호 확인 | deltaTable.history().select("version", "timestamp") |
| 시간 기준 복원 | restoreToTimestamp("2025-04-06 15:00:00") 가능 |
| 복원 후 검증 | SELECT COUNT(\*), SELECT \* LIMIT 10 등으로 확인 가능 |