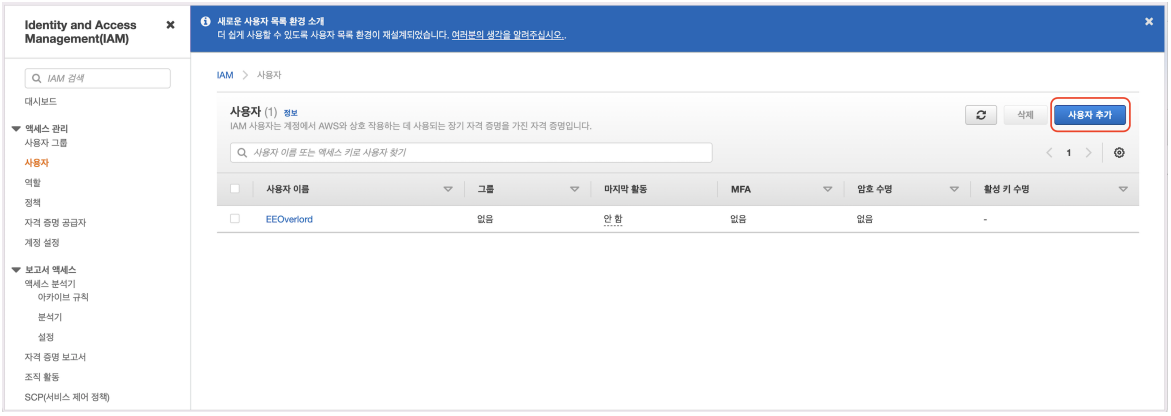
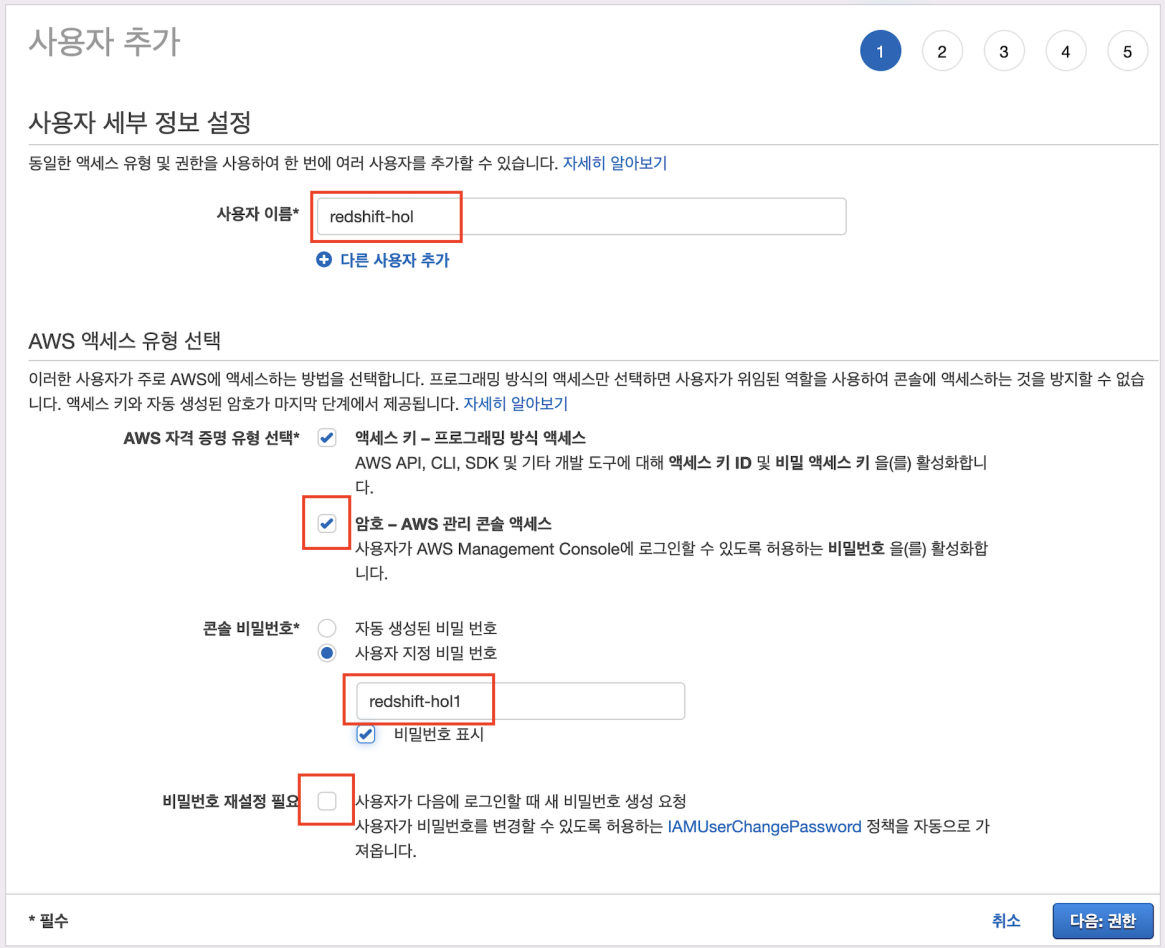
1️⃣ Lab 1

0. Redshift IAM 사용자 추가

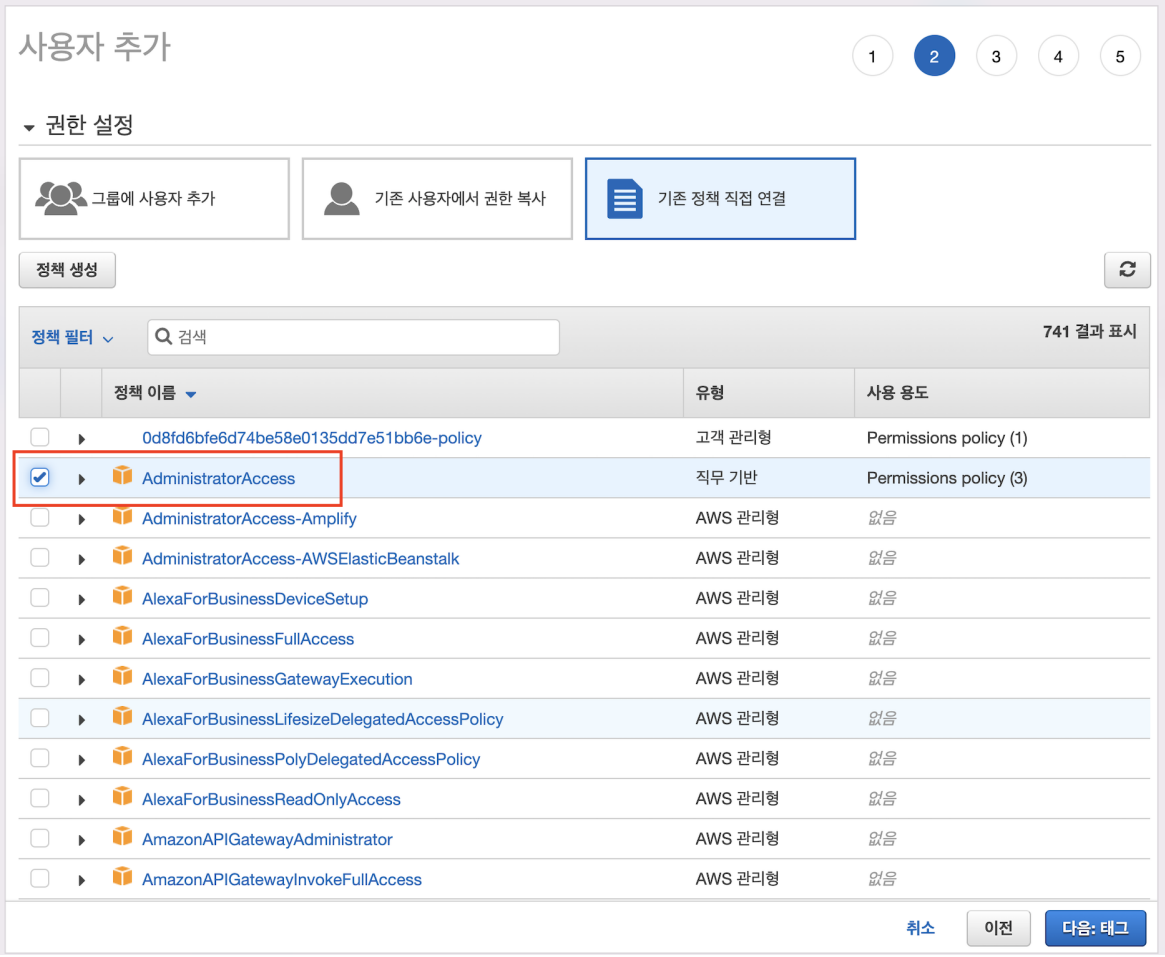
1. 콘솔 접속
   - https://us-east-1.console.aws.amazon.com/iamv2/home?region=us-west-2#/users
     - 또는, https://console.aws.amazon.com/iamv2/home#/users



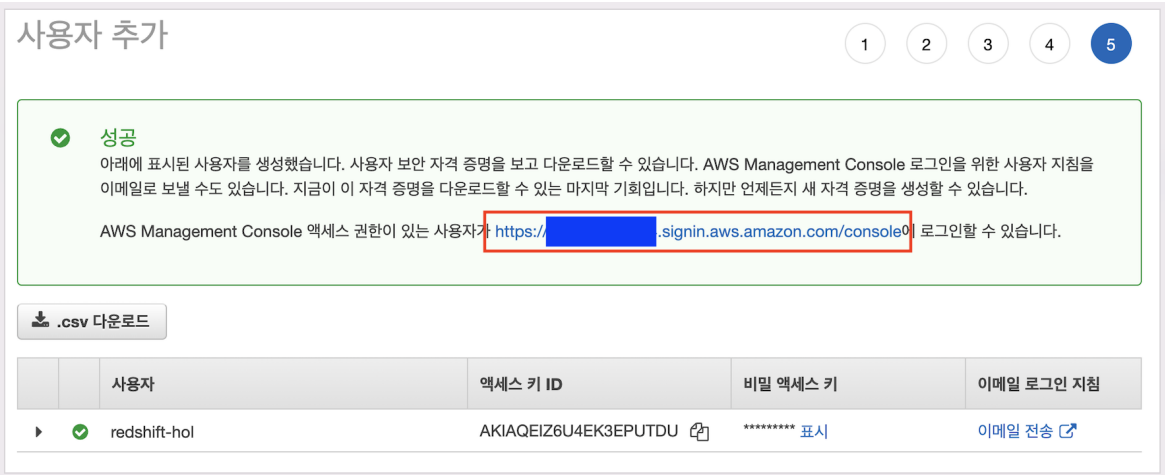
2. 사용자 추가
   - 사용자 이름: `redshift-hol`
   - 콘솔 비밀번호: `redshift-hol1`



3. 권한 설정
   - 정책 이름 `AdministratorAccess` 선택



4. 검토 및 완료
   - 테그 추가(선택 사항)에서 **다음:검토** 버튼을 클릭



5. 로그 아웃하고, 새로 생성한 `redshift-hol` 사용자 로그인

1. 클러스터 생성 및 연결

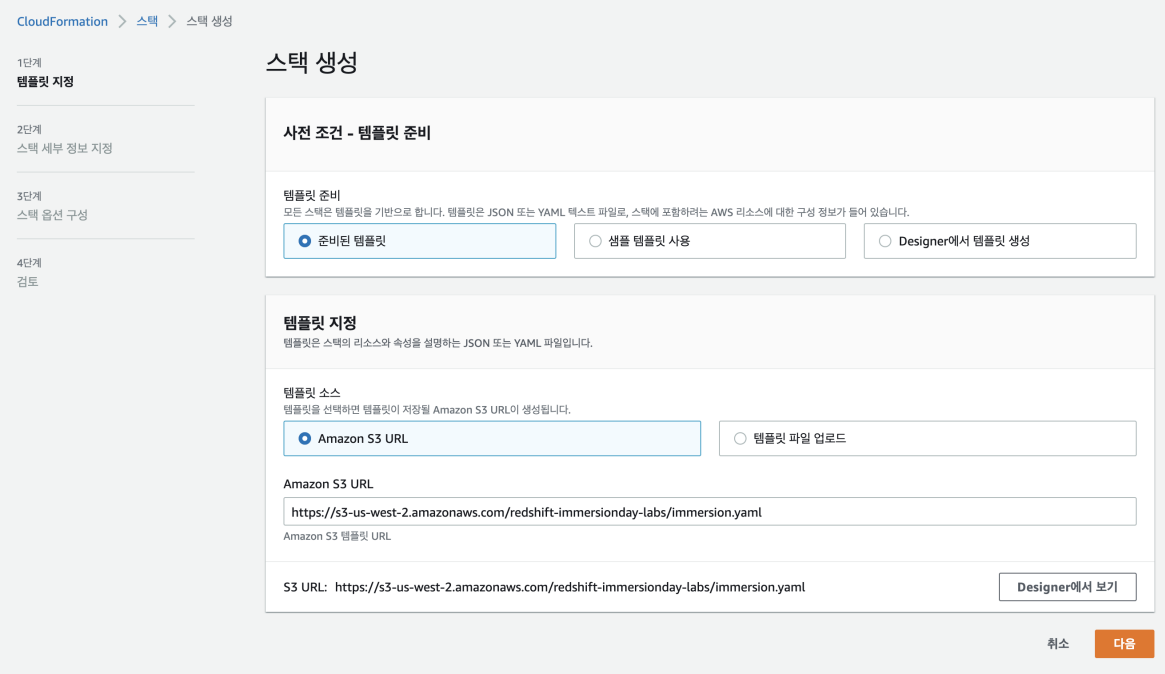
시작에 앞서

- AWS Console 에 로그인 하세요. 이번 Lab에서는 아래 두 가지 정보가 필요합니다.
  - [Your-AWS_Account_Id]
  - [Your_AWS_User_Name]
- 리전(Region) 선택
  - *US-WEST-2 (Oregon)*
  - 이벤트 엔진이 해당 리전을 기준으로 설정이 되어 있습니다.

CloudForamtion

- CloudFormation 템플릿을 이용하여 Amazon Redshift 클러스터를 설치합니다.
  - 템플릿 **링크**: **Launch Stack 클릭**

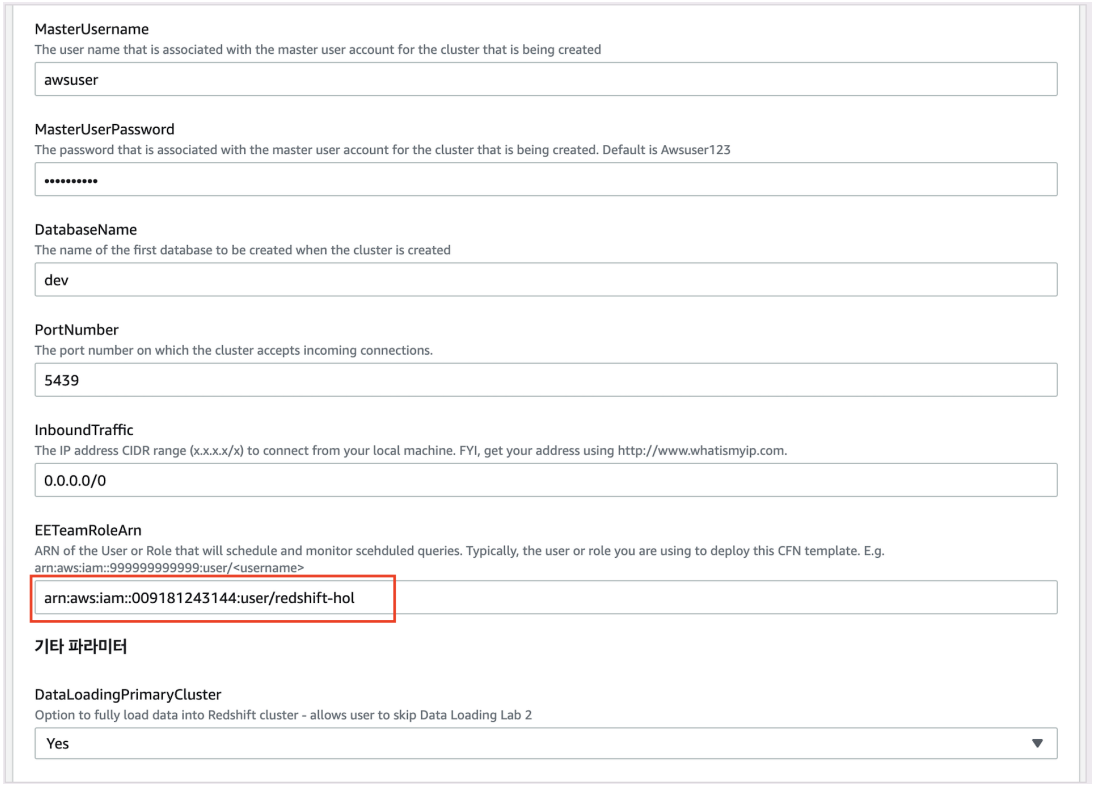
1. 템플릿 지정
   - [다음] 버튼을 클릭하여 `2단계 스택 세부 정보 지정`으로 이동합니다.



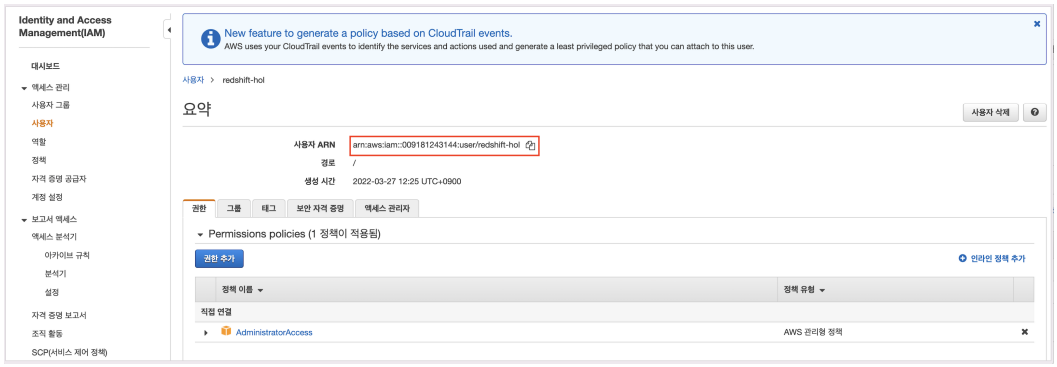
2. 스택 세부 정보 지정
   - 템플릿은 아래와 같은 특징으로 구성되어 있습니다.
     - 네트워크 구성 : 1 VPC, 2 Subnets, 1 Gateway, 1 Security Group
     - 입력 파리미터
       - EETeamRoleArn

```
arn:aws:iam::[Your-AWS_Account_Id]:user/redshift-hol
```

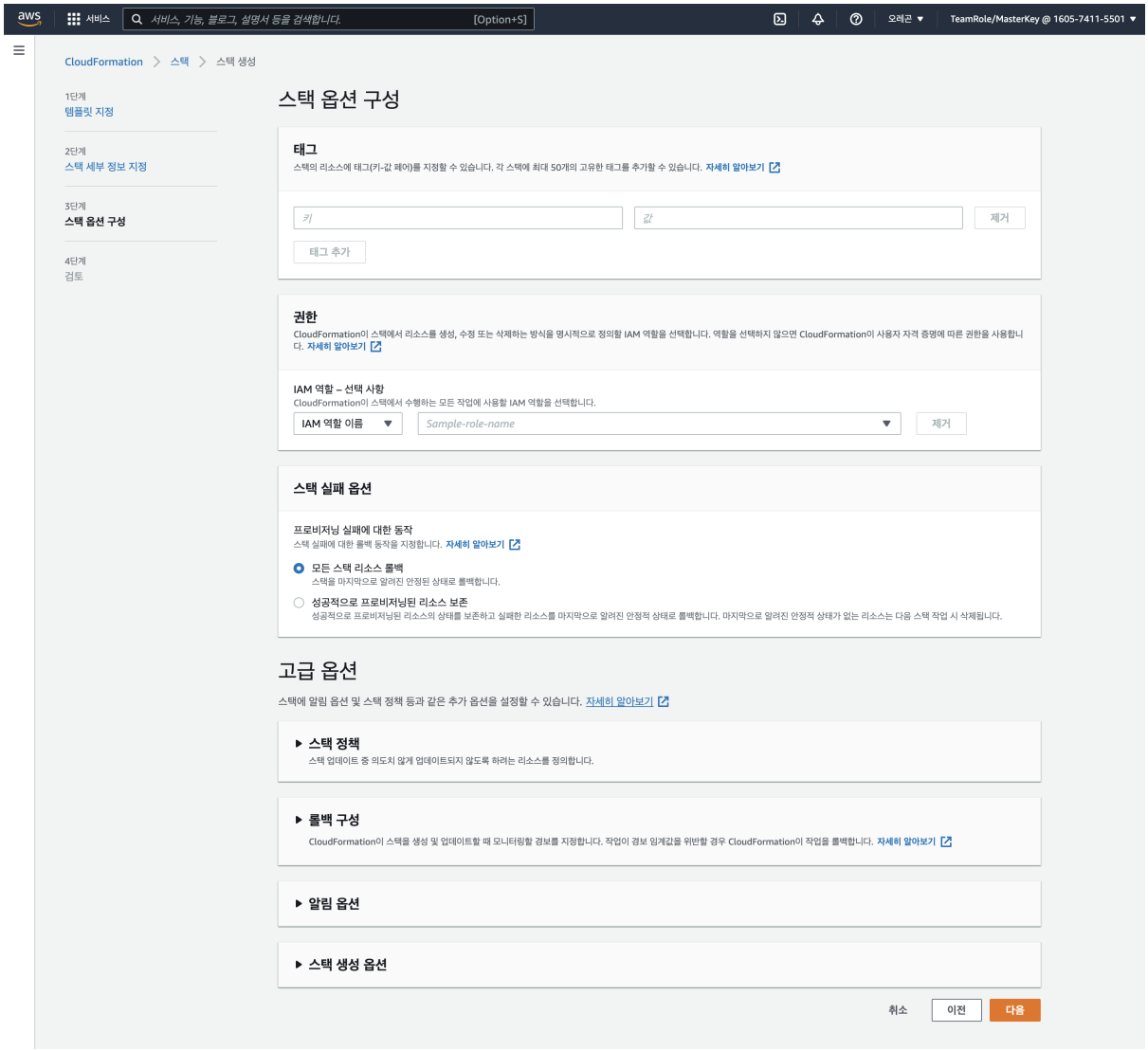
       - 나머지 Default 유지



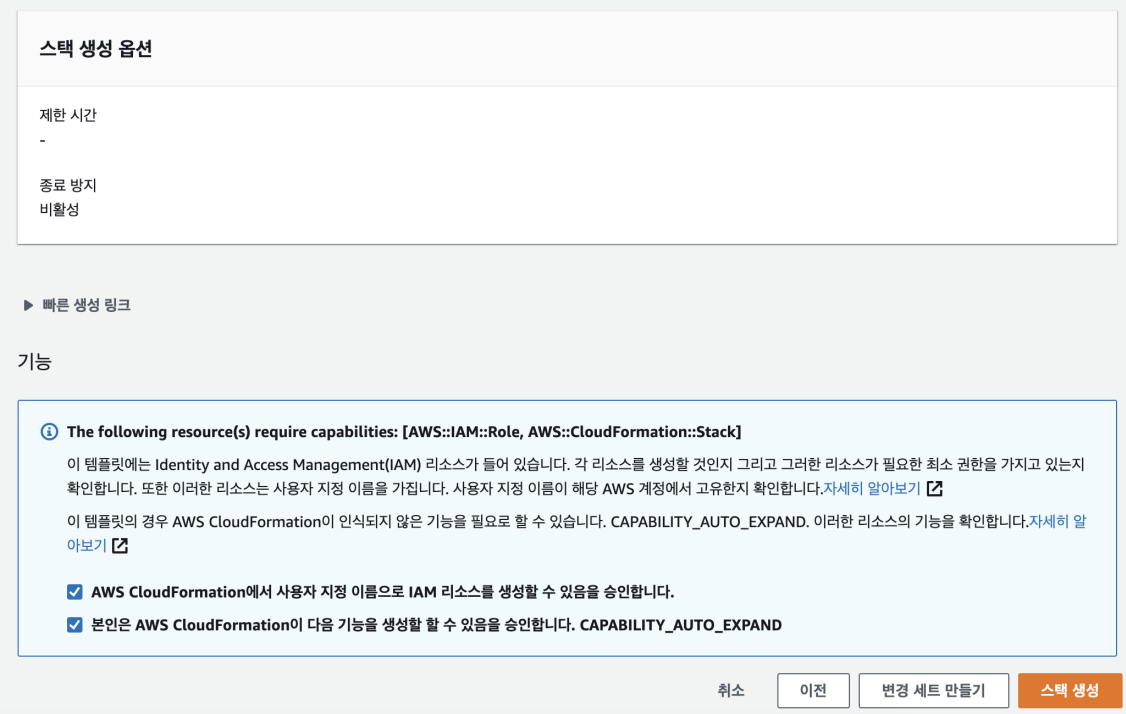
- EETeamRoleArn 확인 방법
  - IAM - User에서 `사용자 ARN` 복사



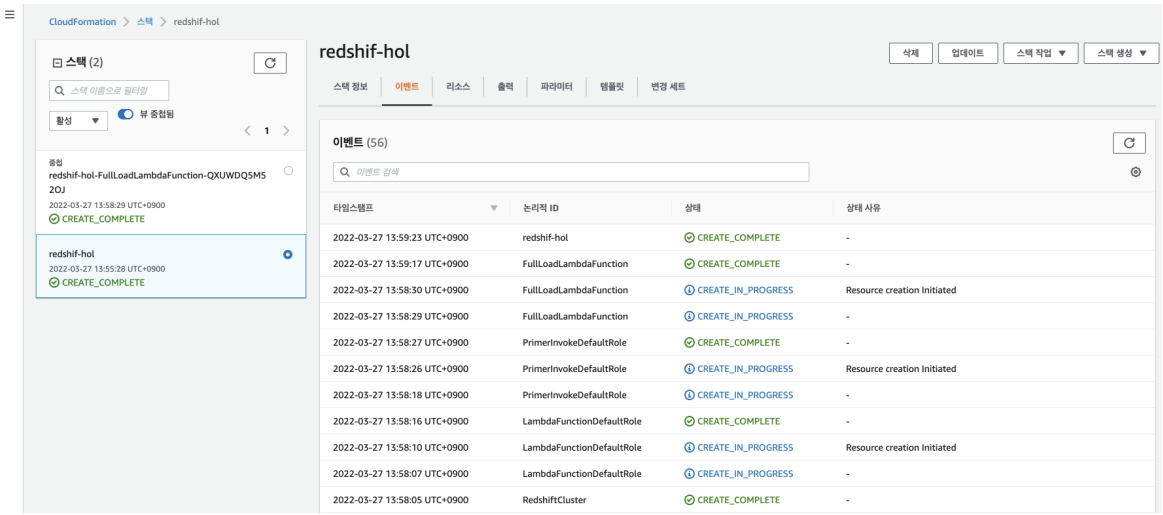
3. 스택 옵션 구성
   - [다음] 버튼을 클릭하여 `4단계 검토`로 이동합니다.



4. 검토
    - 최하단의 기능 부분에서 아래와 같이 체크박스를 확인한 후 [스택 생성]을 클릭합니다.



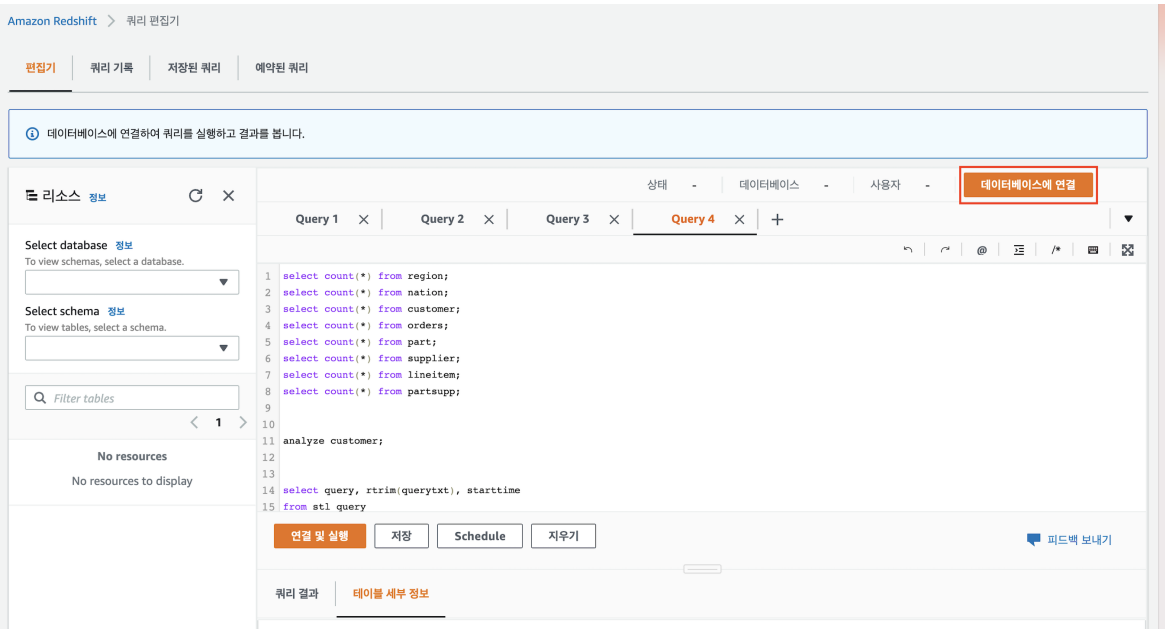
5. 결과 확인



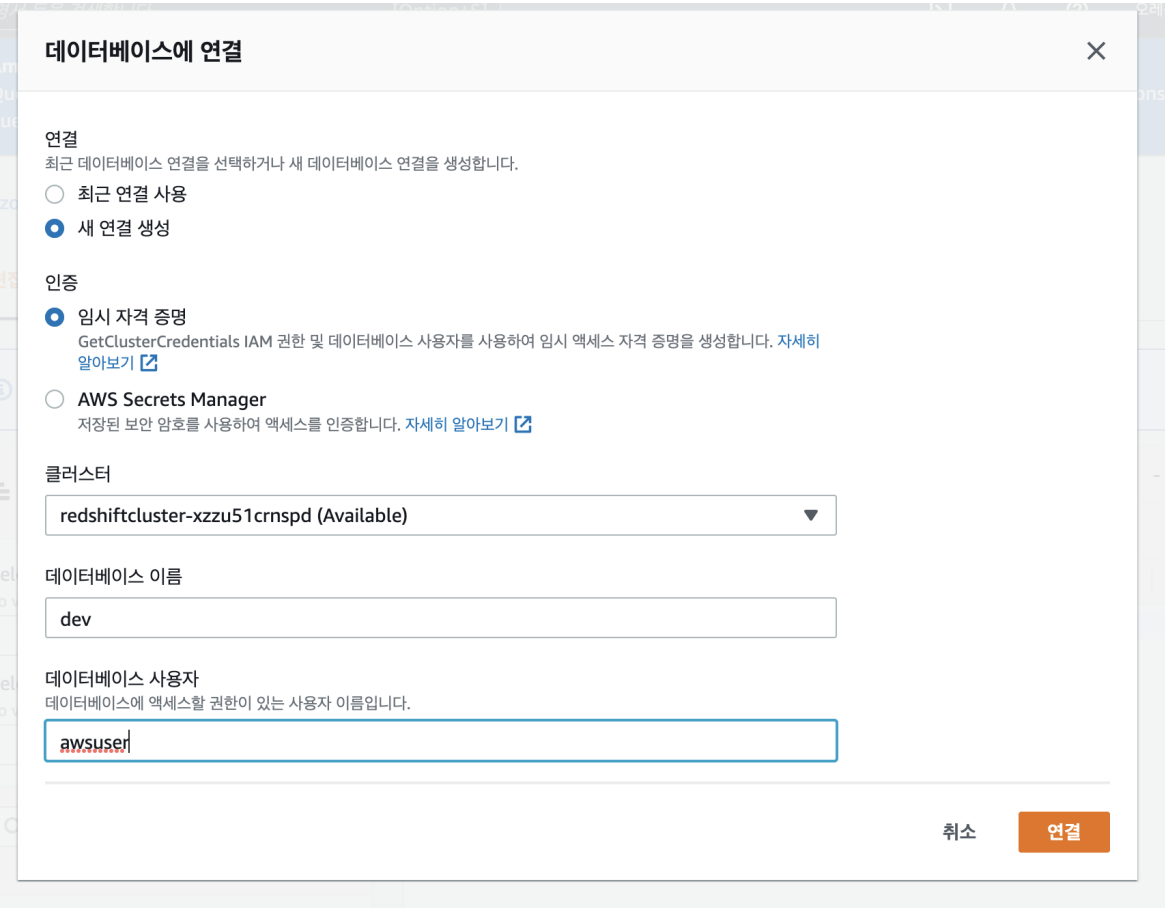
Client Tool - Query Editor V1

이번 섹션에서는 웹기반 쿼리에디터 Query editor v1 을 활용해 보겠습니다.

- 데이터 베이스 연결
  - *데이터베이스에 연결* 버튼 클릭



- 입력항목
  - 연결 → `새 연결 생성`
  - 인증 → `임시 자격 증명`
  - Cluster → `redshiftcluster-xxxxxxxxxxx(Available)`
  - Database name → `dev`
  - Database User → `awsuser`

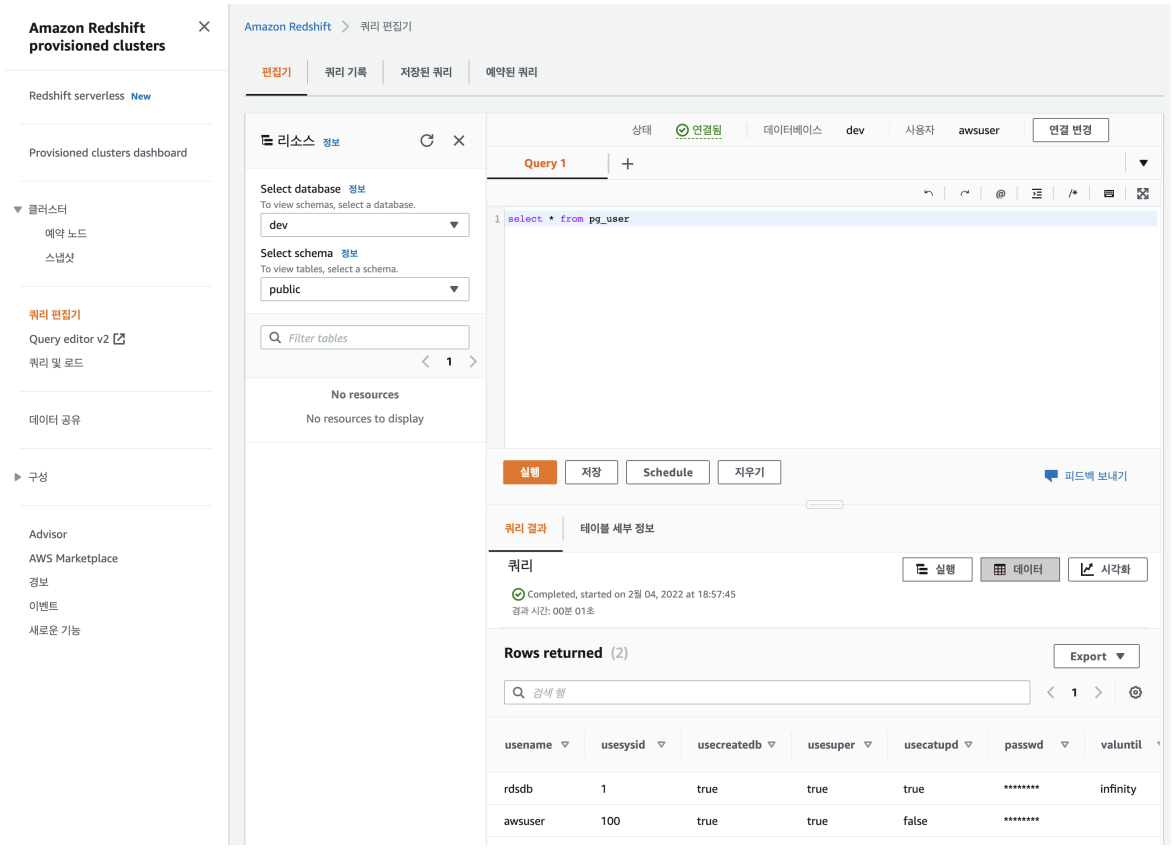


샘플쿼리 실행

- 아래 쿼리를 실행하여 Redshift 클러스터 사용자 정보를 조회합니다.

```
select * from pg_user
```

- 아래와 같이 결과가 출력된다면, 연결이 정상적으로 설정된 것입니다.



2. 클러스터 생성 및 연결

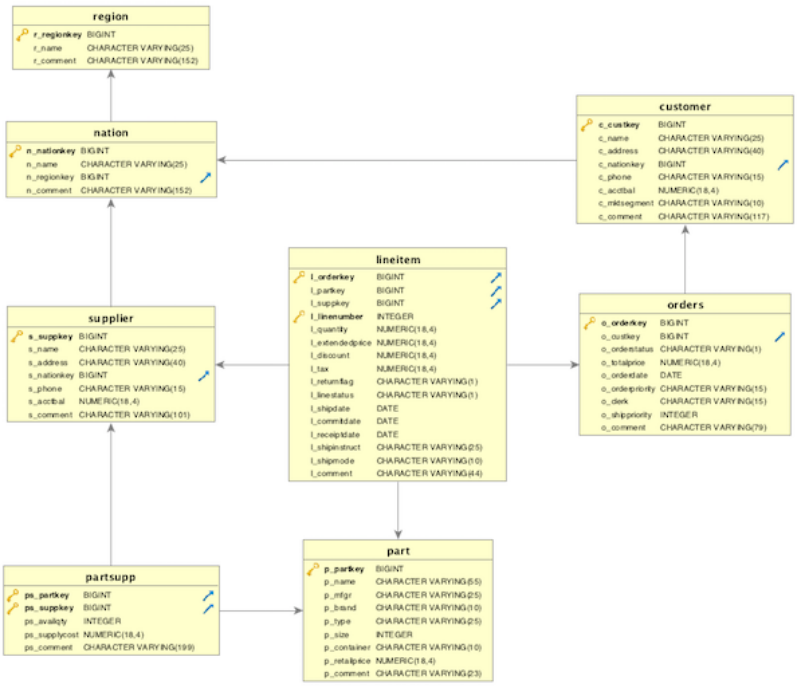
이번 Lab에서는 TPC 벤치마크 모델을 이용하여 8개의 테이블 데이터를 업로드합니다. 관련된 테이블을 Redshift 클러스터에 생성하고, S3에 저장된 샘플 데이터를 업로드합니다.

시작에 앞서

- 이번 Lab을 수행하기 위해서 Redshift 클러스터와 클라이언트 도구 설정이 필요합니다. `Lab 1 - 클러스터 생성 및 연결` 이 먼저 선행되어야 합니다.
- 만약 Lab 1 에서 CFN 템플릿의 `DataLoadingPrimaryCluster` 파라미터를 `Yes` 로 지정하였다면, 이미 모든 테이블이 생성되고 데이터 로딩이 실행된 것입니다. 따라서 아래 *테이블 생성 및 데이터 로딩* 섹션을 수행하실 필요가 없습니다.

테이블 생성

아래 스크립트를 활용하여 TPC 벤치마크 데이터 모델을 위한 테이블을 생성합니다.




```
DROP TABLE IF EXISTS partsupp;
DROP TABLE IF EXISTS lineitem;
DROP TABLE IF EXISTS supplier;
DROP TABLE IF EXISTS part;
DROP TABLE IF EXISTS orders;
DROP TABLE IF EXISTS customer;
DROP TABLE IF EXISTS nation;
DROP TABLE IF EXISTS region;

CREATE TABLE region (
  R_REGIONKEY bigint NOT NULL,
  R_NAME varchar(25),
  R_COMMENT varchar(152))
diststyle all;

CREATE TABLE nation (
  N_NATIONKEY bigint NOT NULL,
  N_NAME varchar(25),
  N_REGIONKEY bigint,
  N_COMMENT varchar(152))
diststyle all;

create table customer (
  C_CUSTKEY bigint NOT NULL,
  C_NAME varchar(25),
  C_ADDRESS varchar(40),
  C_NATIONKEY bigint,
  C_PHONE varchar(15),
  C_ACCTBAL decimal(18,4),
  C_MKTSEGMENT varchar(10),
  C_COMMENT varchar(117))
diststyle all;

create table orders (
  O_ORDERKEY bigint NOT NULL,
  O_CUSTKEY bigint,
  O_ORDERSTATUS varchar(1),
  O_TOTALPRICE decimal(18,4),
  O_ORDERDATE Date,
  O_ORDERPRIORITY varchar(15),
  O_CLERK varchar(15),
  O_SHIPPRIORITY Integer,
  O_COMMENT varchar(79))
distkey (O_ORDERKEY)
sortkey (O_ORDERDATE);

create table part (
  P_PARTKEY bigint NOT NULL,
  P_NAME varchar(55),
  P_MFGR  varchar(25),
  P_BRAND varchar(10),
```

```
  P_TYPE varchar(25),
  P_SIZE integer,
  P_CONTAINER varchar(10),
  P_RETAILPRICE decimal(18,4),
  P_COMMENT varchar(23))
diststyle all;

create table supplier (
  S_SUPPKEY bigint NOT NULL,
  S_NAME varchar(25),
  S_ADDRESS varchar(40),
  S_NATIONKEY bigint,
  S_PHONE varchar(15),
  S_ACCTBAL decimal(18,4),
  S_COMMENT varchar(101))
diststyle all;

create table lineitem (
  L_ORDERKEY bigint NOT NULL,
  L_PARTKEY bigint,
  L_SUPPKEY bigint,
  L_LINENUMBER integer NOT NULL,
  L_QUANTITY decimal(18,4),
  L_EXTENDEDPRICE decimal(18,4),
  L_DISCOUNT decimal(18,4),
  L_TAX decimal(18,4),
  L_RETURNFLAG varchar(1),
  L_LINESTATUS varchar(1),
  L_SHIPDATE date,
  L_COMMITDATE date,
  L_RECEIPTDATE date,
  L_SHIPINSTRUCT varchar(25),
  L_SHIPMODE varchar(10),
  L_COMMENT varchar(44))
distkey (L_ORDERKEY)
sortkey (L_RECEIPTDATE);

create table partsupp (
  PS_PARTKEY bigint NOT NULL,
  PS_SUPPKEY bigint NOT NULL,
  PS_AVAILQTY integer,
  PS_SUPPLYCOST decimal(18,4),
  PS_COMMENT varchar(199))
diststyle even;
```

데이터 로딩

COPY 명령어를 이용하여 로딩합니다.(INSERT 명령어 대비 대규모 데이터를 효율적으로 처리합니다.)

다수의 파일을 한번의 COPY 명령어로 처리할 수 있으며, Amazon Redshift가 병렬로 동작하여 성능을 높입니다. 편의를 위하여 공용 Amazon S3 버킷을 이용하고, `COMPUPDATE` 파라미터를 지정하여 압축분석이 가능하게 합니다.

```
COPY region FROM 's3://redshift-immersionday-labs/data/region/region.tbl.lzo'
iam_role default
region 'us-west-2' lzop delimiter '|' COMPUPDATE PRESET;

COPY nation FROM 's3://redshift-immersionday-labs/data/nation/nation.tbl.'
iam_role default
region 'us-west-2' lzop delimiter '|' COMPUPDATE PRESET;

copy customer from 's3://redshift-immersionday-labs/data/customer/customer.tbl.'
iam_role default
region 'us-west-2' lzop delimiter '|' COMPUPDATE PRESET;

copy orders from 's3://redshift-immersionday-labs/data/orders/orders.tbl.'
iam_role default
region 'us-west-2' lzop delimiter '|' COMPUPDATE PRESET;

copy part from 's3://redshift-immersionday-labs/data/part/part.tbl.'
iam_role default
region 'us-west-2' lzop delimiter '|' COMPUPDATE PRESET;

copy supplier from 's3://redshift-immersionday-labs/data/supplier/supplier.json' manifest
iam_role default
region 'us-west-2' lzop delimiter '|' COMPUPDATE PRESET;

copy lineitem from 's3://redshift-immersionday-labs/data/lineitem-part/'
iam_role default
region 'us-west-2' gzip delimiter '|' COMPUPDATE PRESET;

copy partsupp from 's3://redshift-immersionday-labs/data/partsupp/partsupp.tbl.'
```

```
iam_role default
region 'us-west-2' lzop delimiter '|' COMPUPDATE PRESET;
```

> 1. COMPUPDATE PRESET ON : 테이블 데이터의 분석없이 Amazon Redshift 모범 사례를 활용하여 컬럼의 데이터 타입에 압축을 적용한다.
> 2. REGION 테이블은 특정 파일(region.tbl.lzo)을 가리키고, 나머지 테이블은 prefix를 활용하여 여러 파일(lineitem.tbl.)을 참조한다.
> 3. SUPPLIER 테이블은 JSON 형태의 파일을 참조한다.

💡 예상 소요시간(ra3.xlplus 2개 노드 사용시)

REGION (5 rows) - 20s
NATION (25 rows) - 20s
CUSTOMER (15M rows) – 3m
ORDERS - (76M rows) - 1m
PART - (20M rows) - 4m
SUPPLIER - (1M rows) - 1m
LINEITEM - (303M rows) - 13m
PARTSUPPLIER - (80M rows) 3m

- 데이터 로딩 확인

```
select count(*) from region;
select count(*) from nation;
select count(*) from customer;
select count(*) from orders;
select count(*) from part;
select count(*) from supplier;
select count(*) from lineitem;
select count(*) from partsupp;
```

테이블 관리 - Analyze

쿼리플래너(옵티마이저)가 최적의 실행 계획을 작성할 수 있도록 정기적으로 통계 메타데이터를 업데이트해야 합니다. ANALYZE 명령어를 실행하여 명시적으로 테이블의 통계정보를 생성할 수 있습니다. COPY 명령어로 데이터를 로딩할 때, STATUPDATE를 ON을 셋팅해 주면 데이터가 자동적으로 증분하여 로딩되면서 통계 분석을 생성합니다. 비어있는 테이블에 데이터가 로딩이 되면, COPY 명령어는 기본적으로 ANALYZE를 수행합니다.

예제로 CUSTOMER 테이블에 ANALYZE 명령어를 실행합니다.

```
analyze customer;
```

STL_QUERY 또는 STV_STATEMENTTEXT 테이블을 활용하여 ANALYZE 실행된 시간을 확인할 수 있습니다. 아래 예제 쿼리를 실행하여 CUSTOME 테이블에 최종 통계정보가 생성된 시간을 확인해 보겠습니다.

```
select query, rtrim(querytxt), starttime
from stl_query
where
querytxt like 'padb_fetch_sample%' and
querytxt like '%customer%'
order by query desc
```

> ANALYZE 명령이 실행될 때 Amazon Redshift는 `padb_fetch_sample`과 비슷한 쿼리를 여러번 실행합니다. ANALYZE의 타임스탬프 정보는 COPY 명령이 실행된 시간과 상관 관계가 있으며 Redshift는 테이블에서

> 변경된 데이터가 없으면 ANALYZE 작업을 실행할 필요가 없다는 것을 알고 있습니다.

테이블 관리 - VACUUM

삭제 및 수정이 빈번하게 발생했을 경우 VACUUM 명령어를 수행합니다. Amazon Redshift의 UPDATE 기본 동장은 행을 삭제하고 새로운 행을 삽입하는 절차를 가집니다. Amazon Redshift의 경우 VACUUM을 자동적으로 수행해주는 기능이 추가 되었지만, 수행 방법을 숙지하여 수동적으로 진행할 수도 있습니다. 특정 조건에 따라 전체, 삭제, 정렬을 구분하여 VACUUM 명령어를 활용할 수 있습니다.

1. ORDERS 테이블의 초기 적재상태 확인

```
select col, count(*)
from stv_blocklist, stv_tbl_perm
where stv_blocklist.tbl = stv_tbl_perm.id and
stv_blocklist.slice = stv_tbl_perm.slice and
stv_tbl_perm.name = 'orders' and
col <= 5
group by col
order by col;
```

col	count
0	264
1	248
2	24
3	304
4	312
5	208

2. ORDERS 테이블 일부 행 삭제

```
delete orders where o_orderdate between '1997-01-01' and '1998-01-01';
```

3. Redshift가 자동적으로 삭제 공간을 정리하지 않은 것을 확인

```
select col, count(*)
from stv_blocklist, stv_tbl_perm
where stv_blocklist.tbl = stv_tbl_perm.id
and stv_blocklist.slice = stv_tbl_perm.slice
and stv_tbl_perm.name = 'orders' and
col <= 5
group by col
order by col;
```

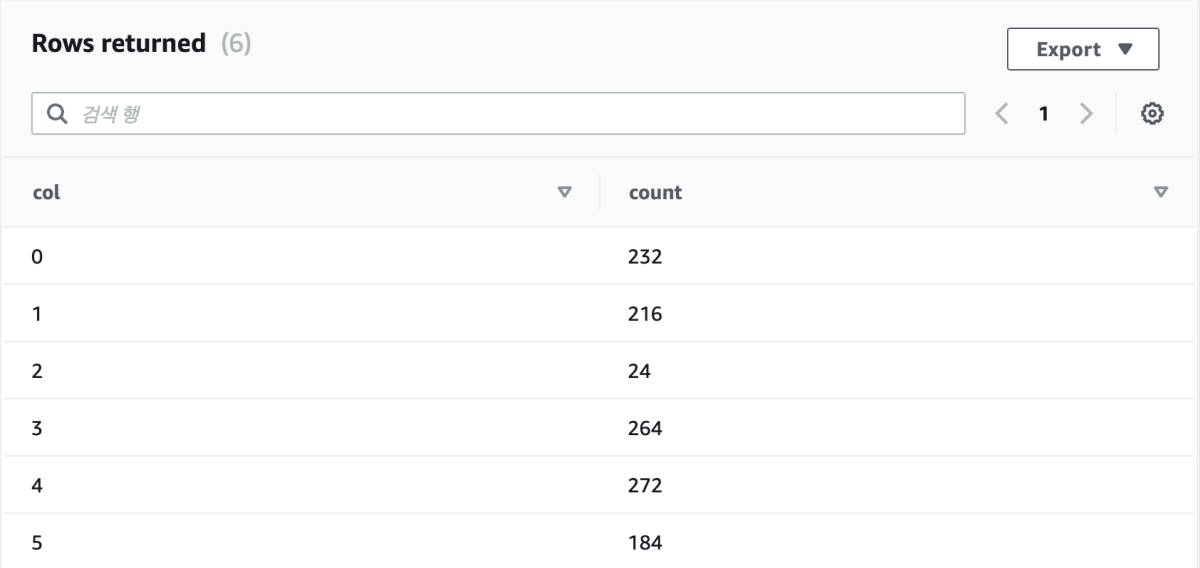
4. VACUUM 명령어 실행

```
vacuum delete only orders;
```

5. 다음 쿼리를 실행하여 VACUUM 명령어를 통하여 공간이 회수된 것을 확인

```
select col, count(*)
from stv_blocklist, stv_tbl_perm
where stv_blocklist.tbl = stv_tbl_perm.id
and stv_blocklist.slice = stv_tbl_perm.slice
and stv_tbl_perm.name = 'orders' and
col <= 5
```

```
group by col
order by col;
```



Rows returned (6)

Export

검색 행

1

col	count
0	232
1	216
2	24
3	264
4	272
5	184

> 열이 매우 적고 행이 많을 경우, 숨겨진 3개의 메타데이터(INSERT_XID, DELETE_XID, ROW_ID)가 테이블의 디스크 공간을 과도하게 사용할 수 있습니다. 숨겨진 메타데이터의 압축 효율을 높이기 위해서, 가능하면 단일 COPY를 이용하여 데이터를 로딩하세요. 여러개의 COPY명령어를 이용할 경우 INSERT_XID 컬럼은 압축 효율이 좋지않을 수도 있습니다. 여러번 VACUUM을 실행해도 압축 효율이 좋아지지 않을 수도 있습니다.
> https://docs.aws.amazon.com/ko_kr/ko_kr/redshift/latest/dg/c_load_compression_hidden_cols.html

트러블 슈팅

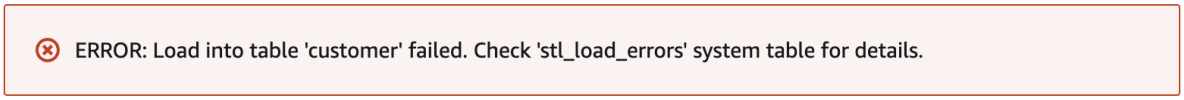
아래 시스템 테이블을 이용하여 데이터 로딩과 관련된 트러블 슈팅을 합니다

- STL_LOAD_ERRORS
- STL_FILE_SCAN

추가로, 실제로 데이터를 적재하지 않아도 검증하는 방법이 있습니다. NOLOAD옵션을 이용하여 실제 적재가 발생하기 전에 데이터 파일이 COPY 명령어 수행 시 오류가 발생하는지 여부를 확인할 수 있습니다. NOLOAD 옵션의 경우 파일 파싱만을 하기 때문에 빠르게 수행됩니다.

아래 CUSTOMER 테이블을 활용하여 컬럼이 미일치 하는 예제를 수행해 봅니다.

```
COPY customer FROM 's3://redshift-immersionday-labs/data/nation/nation.tbl.'
iam_role default
region 'us-west-2' lzop delimiter '|' noload;
```



ERROR: Load into table 'customer' failed. Check 'stl_load_errors' system table for details.

```
ERROR: Load into table 'customer' failed.  Check 'stl_load_errors' system table for details. [SQL State=XX000]
```

STL_LOAD_ERROR 시스템 테이블을 사용하여 오류 사항을 확인합니다.

```
select * from stl_load_errors;
```

STL_LOAD_ERRORS과 STV_TBL_PERM을 JOIN하여 테이블 이름을 기준으로 일치하는 테이블 ID 활용하여 오류 정보를 추출할 수 있습니다.

```
create view loadview as
(select distinct tbl, trim(name) as table_name, query, starttime,
trim(filename) as input, line_number, colname, err_code,
trim(err_reason) as reason
from stl_load_errors sl, stv_tbl_perm sp
where sl.tbl = sp.id);

-- Query the LOADVIEW view to isolate the problem.
select * from loadview where table_name='customer';
```

3. 테이블 디자인 및 쿼리 튜닝

이번 Lab에서는 Redshift의 압축, 비정규화, 분산처리, 정렬에 따른 영향도에 따른 쿼리 성능을 분석합니다.

시작에 앞서

- 이번 Lab을 수행하기 위해서 Redshift 클러스터와 클라이언트 도구 설정이 필요합니다. `Lab 1 - 클러스터 생성 및 연결`, `Lab2 - Data Loading`이 먼저 선행되어야 합니다.
- 이번 Lab 수행을 위해서는 아래 정보가 필요합니다
  - [Your-Redshift_User]: `awsuser`

결과셋 캐싱과 실행계획 재사용

Redshift는 캐싱된 결과셋을 활용하여 검색속도를 높일 수 있습니다. 만약 테이블의 기본정보가 변경되지 않았다면 캐싱된 데이터를 이용할 수 있습니다. 또한, 검색 조건의 일부만 변경될 경우 이전에 컴파일된 쿼리 플랜을 재사용하여 검색속도를 높입니다.

1. 다음 쿼리를 실행하고 쿼리 실행 시간을 기록합니다. 첫 번째 실행이기 때문에 Redshift는 쿼리를 컴파일하고 결과셋을 캐시상태로 저장합니다.

```
SELECT c_mktsegment, o_orderpriority, sum(o_totalprice)
FROM customer c
JOIN orders o on c_custkey = o_custkey
GROUP BY c_mktsegment, o_orderpriority;
```

2. 같은 쿼리를 재실행한 후 실행 시간을 다시 기록합니다. 두 번째 실행은 캐시되어있는 결과셋을 이용하기 때문에 즉시 응답을 얻을 수 있습니다.

```
SELECT c_mktsegment, o_orderpriority, sum(o_totalprice)
FROM customer c
JOIN orders o on c_custkey = o_custkey
GROUP BY c_mktsegment, o_orderpriority;
```

3. Customer 테이블에 데이터를 업데이트 하고 쿼리를 재실행합니다. 테이블의 정보가 변경되었을 경우 Redshift는 변경 사항을 인식하고 캐시된 결과셋을 무효처리합니다. 2단계에서 수행했던 결과보다 느리지만 컴파일된 실행 계획을 재사용하기 때문에 1단계 보다는 빠른 것을 확인할 수 있습니다.

```
UPDATE customer
SET c_mktsegment = c_mktsegment
WHERE c_mktsegment = 'MACHINERY';

VACUUM DELETE ONLY customer;

SELECT c_mktsegment, o_orderpriority, sum(o_totalprice)
FROM customer c
JOIN orders o on c_custkey = o_custkey
GROUP BY c_mktsegment, o_orderpriority;
```

4. 조건 절이 있는 새로운 쿼리를 실행합니다. 첫 실행이기 때문에 Redshift는 쿼리를 컴파일하고 결과셋을 캐시 저장합니다.

```
SELECT c_mktsegment, count(1)
FROM Customer c
WHERE c_mktsegment = 'MACHINERY'
GROUP BY c_mktsegment;
```

5. 검색 조건에 변화를 주어 비슷한 양의 데이터를 스캔 및 집계하여도 이전 실행보다 빠른 것을 확인할 수 있습니다. 검색 조건의 검색어만 변경되었기 때문에 컴파일된 실행 계획을 재사용합니다. 다양한 사용자가 검색조건만 다르게 검색하는 쿼리는 BI 리포팅에서 빈번하게 활용되는 유형입니다.

```
SELECT c_mktsegment, count(1)
FROM customer c
WHERE c_mktsegment = 'BUILDING'
GROUP BY c_mktsegment;
```

6. 이번 실습의 나머지 부분에서는 애드혹 쿼리에 대한 실행시간을 확인할 수 있도록 결과셋 캐싱 기능을 비활성화 합니다.

```
ALTER USER awsuser set enable_result_cache_for_session to false;
```

선택적 필터링

Redshift는 특정 필터 기준과 일치하지 않는 데이터 블록에 대해서 옵티마이저가 읽기를 스킵하여 속도를 높일 수 있습니다. 아래 예제에서 Orders 테이블의 경우 o_order_date 정렬 키가 정의되어 있기 때문에, 해당 컬럼을 활용할 경우 쿼리 속도가 빨라집니다.

1. 아래 쿼리를 2회 실행하고 수행시간을 확인합니다. 첫 번째 실행은 쿼리를 컴파일하여 실행계획을 생성하지만, 두 번째 실행에서는 더욱 빠른 결과를 얻어낼 수 있습니다.

```
SELECT count(1), sum(o_totalprice)
FROM orders
WHERE o_orderdate between '1992-07-05' and '1992-07-07';

SELECT count(1), sum(o_totalprice)
FROM orders
WHERE o_orderdate between '1992-07-05' and '1992-07-07';
```

2. 두 번째 실행의 실행 시간을 기록하면서 다음 쿼리를 두 번 실행합니다. 다시 말하지만, 첫 번째 쿼리는 계획이 컴파일되었는지 확인하는 것입니다.
    - 참고: 이 쿼리는 이전 단계에서 사용한 쿼리와 다른 필터 조건을 가지고 있지만 상대적으로 동일한 수의 데이터 행을 스캔합니다.

```
SELECT count(1), sum(o_totalprice)
FROM orders
where o_orderkey < 600001;

SELECT count(1), sum(o_totalprice)
FROM orders
where o_orderkey < 600001;
```

3. 다음을 실행하여 각 쿼리의 실행 시간을 비교합니다. 집계된 행 수가 비슷하더라도 두 번째 쿼리가 이전 단계의 쿼리보다 더 오래 걸린다는 것을 알 수 있습니다. 이는 테이블에 정의된 정렬 키(o_orderdate)를 활용하는 첫 번째 쿼리의 기능 때문입니다.

```
SELECT query, TRIM(querytxt) as SQL, starttime, endtime, DATEDIFF(microsecs, starttime, endtime) AS duration
FROM STL_QUERY
WHERE TRIM(querytxt) like '%orders%'
ORDER BY starttime DESC
LIMIT 4;
```

압축

Redshift는 대규모 데이터셋을 활용합니다. Redshift 워크로드를 최적화하기 위한 핵심 원칙 중 하나는 저장된 데이터의 양을 줄이는 것입니다. Redshift는 다양한 유형과 값을 포함하는 데이터 행에서 작업하는 대신 열 방식으로 작동합니다. 이를 통해 단일 데이터 열에서 작동할 수 있는 독립적인 압축 알고리즘을 적용할 수 있는 기회를 제공합니다.

1. `LAB 2 - 데이터 로딩`에서 특정 압축 인코딩 없이 lineitem 테이블이 정의되었습니다. 대신 COPY 문에서 COMPUPDATE PRESET 절을 사용했기 때문에 데이터가 로드될 때 인코딩이 기본값으로 자동으로 적용되었습니다. 다음 쿼리를 실행하여 lineitem 테이블에 사용된 압축을 확인합니다.

```
SELECT tablename, "column", encoding
FROM pg_table_def
WHERE schemaname = 'public' AND tablename = 'lineitem'
```

tablename	column	encoding
lineitem	l_orderkey	az64
lineitem	l_partkey	az64
lineitem	l_suppkey	az64
lineitem	l_linenumber	az64
lineitem	l_quantity	az64
lineitem	l_extendedprice	az64
lineitem	l_discount	az64
lineitem	l_tax	az64
lineitem	l_returnflag	lzo
lineitem	l_linestatus	lzo

2. 각 열의 ENCODING을 RAW로 설정하는 라인 항목 테이블의 사본을 만들고 해당 테이블에 라인 항목 데이터를 로드합니다.

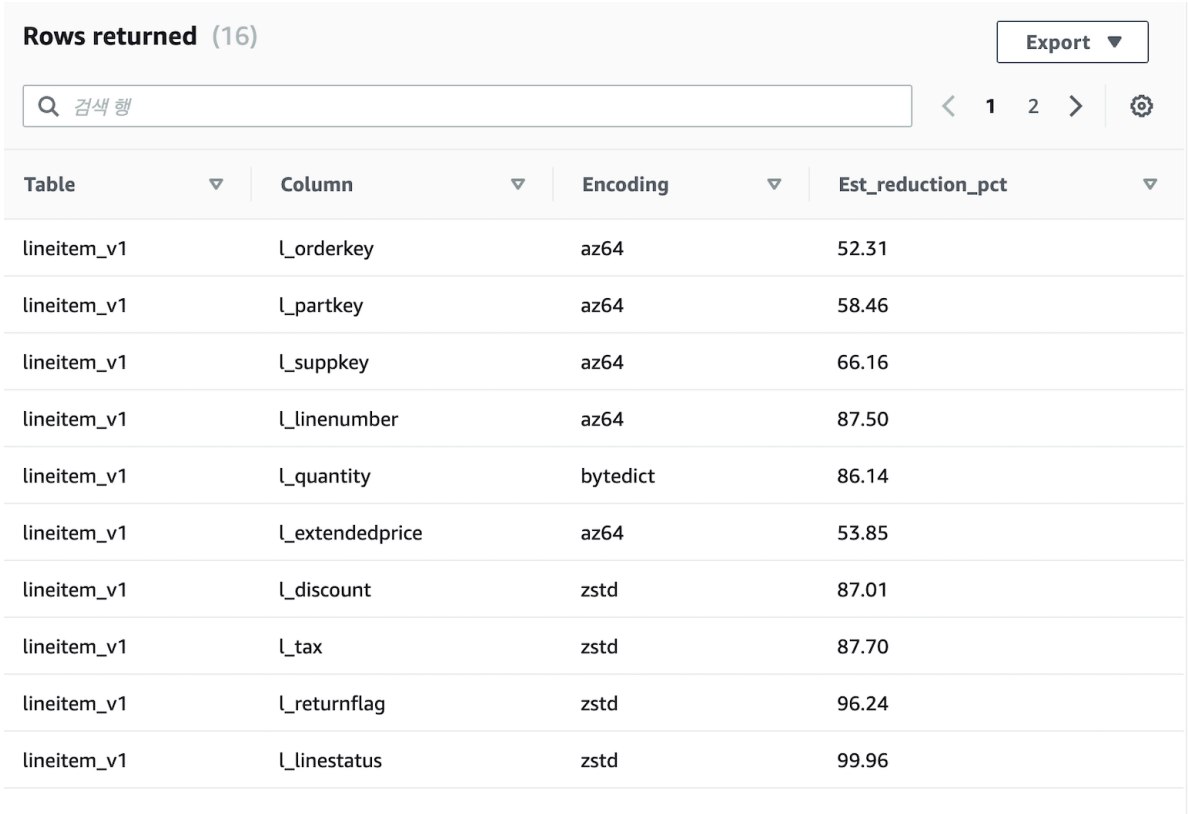
```
DROP TABLE IF EXISTS lineitem_v1;
CREATE TABLE lineitem_v1 (
  L_ORDERKEY bigint NOT NULL ENCODE RAW      ,
  L_PARTKEY bigint ENCODE RAW                ,
  L_SUPPKEY bigint ENCODE RAW                ,
  L_LINENUMBER integer NOT NULL ENCODE RAW   ,
  L_QUANTITY decimal(18,4) ENCODE RAW        ,
  L_EXTENDEDPRICE decimal(18,4) ENCODE RAW   ,
  L_DISCOUNT decimal(18,4) ENCODE RAW        ,
  L_TAX decimal(18,4) ENCODE RAW             ,
  L_RETURNFLAG varchar(1) ENCODE RAW         ,
  L_LINESTATUS varchar(1) ENCODE RAW         ,
  L_SHIPDATE date ENCODE RAW                 ,
  L_COMMITDATE date ENCODE RAW               ,
  L_RECEIPTDATE date ENCODE RAW              ,
  L_SHIPINSTRUCT varchar(25) ENCODE RAW      ,
  L_SHIPMODE varchar(10) ENCODE RAW          ,
  L_COMMENT varchar(44) ENCODE RAW
)
distkey (L_ORDERKEY)
sortkey (L_RECEIPTDATE);

INSERT INTO lineitem_v1
SELECT * FROM lineitem;

ANALYZE lineitem_v1;
```

3. Redshift는 ANALYZE COMPRESSION 명령을 제공합니다. 이 명령은 가장 압축률이 높은 컬럼의 인코딩을 결정합니다. 방금 로드된 테이블에서 ANALYZE COMPRESSION 명령을 실행합니다. 결과를 기록하고 1단계의 결과와 비교합니다.

```
ANALYZE COMPRESSION lineitem_v1;
```



Rows returned (16)

Export

검색 행

1 2

Table	Column	Encoding	Est_reduction_pct
lineitem_v1	l_orderkey	az64	52.31
lineitem_v1	l_partkey	az64	58.46
lineitem_v1	l_suppkey	az64	66.16
lineitem_v1	l_linenumber	az64	87.50
lineitem_v1	l_quantity	bytedict	86.14
lineitem_v1	l_extendedprice	az64	53.85
lineitem_v1	l_discount	zstd	87.01
lineitem_v1	l_tax	zstd	87.70
lineitem_v1	l_returnflag	zstd	96.24
lineitem_v1	l_linestatus	zstd	99.96

> 참고: 대부분의 열은 인코딩이 동일하지만 인코딩이 변경되면 일부 열은 압축 성능이 향상됩니다.

4. 압축 여부에 따라 테이블의 저장 공간을 분석합니다. 테이블은 사용된 스토리지 양(MB)을 열 별로 저장합니다. 첫 번째 테이블과 비교하자면 두 번째 테이블의 스토리지가 약 70% 절약된 것을 볼 수 있습니다. 이 쿼리에서는 컬럼당 스토리지 크기와 테이블의 총 스토리지를 제공합니다(각 라인에서 반복되어 출력)

```
SELECT
  CAST(d.attname AS CHAR(50)),
  SUM(CASE WHEN CAST(d.relname AS CHAR(50)) = 'lineitem'
THEN b.size_in_mb ELSE 0 END) AS size_in_mb,
  SUM(CASE WHEN CAST(d.relname AS CHAR(50)) = 'lineitem_v1'
THEN b.size_in_mb ELSE 0 END) AS size_in_mb_v1,
  SUM(SUM(CASE WHEN CAST(d.relname AS CHAR(50)) = 'lineitem'
THEN b.size_in_mb ELSE 0 END)) OVER () AS total_mb,
  SUM(SUM(CASE WHEN CAST(d.relname AS CHAR(50)) = 'lineitem_v1'
THEN b.size_in_mb ELSE 0 END)) OVER () AS total_mb_v1
FROM (
  SELECT relname, attname, attnum - 1 as colid
  FROM pg_class t
  INNER JOIN pg_attribute a ON a.attrelid = t.oid
  WHERE t.relname LIKE 'lineitem%') d
INNER JOIN (
  SELECT name, col, MAX(blocknum) AS size_in_mb
  FROM stv_blocklist b
  INNER JOIN stv_tbl_perm p ON b.tbl=p.id
  GROUP BY name, col) b
ON d.relname = b.name AND d.colid = b.col
GROUP BY d.attname
ORDER BY d.attname;
```

Join 전략

Redshift 분산 아키텍처에서 결합(Join) 데이터를 처리하려면 데이터를 한 노드에서 다른 노드로 브로드캐스트해야 할 수 있습니다. 어떤 조인 전략이 사용 중이고 이를 개선하는 방법이 무엇인지 찾기 위해서 쿼리 실행 계획을 분석하는 것이 중요합니다.

1. 다음 쿼리에 대해 EXPLAIN 명령어를 실행합니다. 이 테이블이 `LAB 2 - 데이터 로드`에서 로딩되었을 때 *customer* 테이블에 대해 DISTSTYLE 옵션이 ALL로 설정되었습니다. ALL 분포는 작은 Dimension 테이블에 대한 좋은 방법입니다. 결과적으로

"DS_DIST_ALL_NONE"의 조인 전략은 상대적으로 낮은 비용이 발생합니다. *order* 및 *lineitem* 테이블의 DISTKEY는 *orderkey* 키입니다. 이 두 테이블은 동일한 키로 분산되어 있으므로 데이터가 함께 배치되고 "DS_DIST_NONE"의 조인 전략을 활용할 수 있습니다.

```
EXPLAIN
SELECT c_mktsegment,COUNT(o_orderkey) AS orders_count, sum(l_quantity) as quantity, sum (l_extendedprice) as extendedprice
FROM lineitem
JOIN orders on l_orderkey = o_orderkey
JOIN customer c on o_custkey = c_custkey
WHERE l_commitdate between '1992-01-01T00:00:00Z' and '1992-12-31T00:00:00Z'
GROUP BY c_mktsegment;
```

```
XN HashAggregate  (cost=77743573.02..77743573.06 rows=5 width=43)
  ->  XN Hash Join DS_DIST_ALL_NONE  (cost=1137500.00..77351933.42 rows=39163960 width=43)
        Hash Cond: ("outer".o_custkey = "inner".c_custkey)
        ->  XN Hash Join DS_DIST_NONE  (cost=950000.00..70800289.92 rows=39163960 width=39)
              Hash Cond: ("outer".l_orderkey = "inner".o_orderkey)
              ->  XN Seq Scan on lineitem  (cost=0.00..8985568.32 rows=79308960 width=31)
                    Filter: ((l_commitdate <= '1992-12-31'::date) AND (l_commitdate >= '1992-01-01'::date))
              ->  XN Hash  (cost=760000.00..760000.00 rows=76000000 width=16)
                    ->  XN Seq Scan on orders  (cost=0.00..760000.00 rows=76000000 width=16)
        ->  XN Hash  (cost=150000.00..150000.00 rows=15000000 width=20)
              ->  XN Seq Scan on customer c  (cost=0.00..150000.00 rows=15000000 width=20)
```

2. 이제 아래 쿼리를 2번 실행하고 실행 시간을 기록합니다. 첫 번째 실행은 계획이 컴파일되었는지 확인하는 것입니다. 두 번째가 최종 사용자 경험을 보다 잘 나타냅니다.

```
SELECT c_mktsegment,COUNT(o_orderkey) AS orders_count, sum(l_quantity) as quantity, sum (l_extendedprice) as extendedprice
FROM lineitem
JOIN orders on l_orderkey = o_orderkey
JOIN customer c on o_custkey = c_custkey
WHERE l_commitdate between '1992-01-01T00:00:00Z' and '1992-12-31T00:00:00Z'
GROUP BY c_mktsegment;
```

3. *custkey*를 사용하여 분산되는 *customer* 테이블의 새 버전을 만듭니다. EXPLAIN을 실행하고 비용이 더 많이 드는 조인 전략 "DS_BCAST_INNER"가 발생하는 것을 확인할 수 있습니다. 이는 *cutomer*과 *order* 테이블이 같은 위치에 있지 않고 내부 테이블의 데이터가 순서대로 조인을 위해서 브로드캐스팅되어야 하기 때문입니다.

```
DROP TABLE IF EXISTS customer_v1;
CREATE TABLE customer_v1
DISTKEY (c_custkey) as
SELECT * FROM customer;

EXPLAIN
SELECT c_mktsegment,COUNT(o_orderkey) AS orders_count, sum(l_quantity) as quantity, sum (l_extendedprice) as extendedprice
FROM lineitem
JOIN orders on l_orderkey = o_orderkey
JOIN customer_v1 c on o_custkey = c_custkey
WHERE l_commitdate between '1992-01-01T00:00:00Z' and '1992-12-31T00:00:00Z'
GROUP BY c_mktsegment;
```

```
XN HashAggregate  (cost=4800043545729.64..4800043545729.68 rows=5 width=44)
  ->  XN Hash Join DS_BCAST_INNER  (cost=4831672.10..4800043141206.92 rows=40452272 width=44)
        Hash Cond: ("outer".o_custkey = "inner".c_custkey)
        ->  XN Hash Join DS_DIST_NONE  (cost=4644172.10..36397557.46 rows=40338536 width=39)
              Hash Cond: ("outer".o_orderkey = "inner".l_orderkey)
              ->  XN Seq Scan on orders  (cost=0.00..760000.00 rows=76000000 width=16)
              ->  XN Hash  (cost=4545123.36..4545123.36 rows=39619498 width=31)
                    ->  XN Seq Scan on lineitem  (cost=0.00..4545123.36 rows=39619498 width=31)
                          Filter: ((l_commitdate <= '1992-12-31'::date) AND (l_commitdate >= '1992-01-01'::date))
        ->  XN Hash  (cost=150000.00..150000.00 rows=15000000 width=21)
              ->  XN Seq Scan on customer_v1 c  (cost=0.00..150000.00 rows=15000000 width=21)
```

4. 이제 아래 쿼리를 2번 실행하고 실행 시간을 기록합니다. 첫 번째 실행은 계획이 컴파일되었는지 확인하는 것입니다. 두 번째가 최종 사용자 경험을 보다 잘 나타냅니다.

```
SELECT c_mktsegment,COUNT(o_orderkey) AS orders_count, sum(l_quantity) as quantity, sum (l_extendedprice) as extendedprice
FROM lineitem
JOIN orders on l_orderkey = o_orderkey
```

```
JOIN customer_v1 c on o_custkey = c_custkey
WHERE l_commitdate between '1992-01-01T00:00:00Z' and '1992-12-31T00:00:00Z'
GROUP BY c_mktsegment;
```

5. 마지막으로 EVEN 분포를 사용하여 배포되는 새 버전의 *orders* 테이블을 만듭니다. EXPLAIN을 실행하여 lineitem 테이블과 동일한 키로 배포되지 않기 때문에 *orders* 테이블에 조인할 때 "DS_DIST_INNER"의 조인 전략이 발생하는지 확인합니다.

```
DROP TABLE IF EXISTS orders_v1;
CREATE TABLE orders_v1
DISTSTYLE EVEN as
SELECT * FROM orders;

EXPLAIN
SELECT c_mktsegment,COUNT(o_orderkey) AS orders_count, sum(l_quantity) as quantity, sum (l_extendedprice) as extendedprice
FROM lineitem
JOIN orders_v1 on l_orderkey = o_orderkey
JOIN customer_v1 c on o_custkey = c_custkey
WHERE l_commitdate between '1992-01-01T00:00:00Z' and '1992-12-31T00:00:00Z'
GROUP BY c_mktsegment;
```

```
XN HashAggregate  (cost=12532460845648.41..12532460845648.45 rows=5 width=44)
  ->  XN Hash Join DS_BCAST_INNER  (cost=992960.75..12532460502002.24 rows=34364617 width=44)
        Hash Cond: ("outer".o_custkey = "inner".c_custkey)
        ->  XN Hash Join DS_DIST_INNER  (cost=805460.75..7732454744986.53 rows=34267997 width=39)
              Inner Dist Key: orders_v1.o_orderkey
              Hash Cond: ("outer".l_orderkey = "inner".o_orderkey)
              ->  XN Seq Scan on lineitem  (cost=0.00..4545123.36 rows=39619498 width=31)
                    Filter: ((l_commitdate <= '1992-12-31'::date) AND (l_commitdate >= '1992-01-01'::date))
              ->  XN Hash  (cost=644368.60..644368.60 rows=64436860 width=16)
                    ->  XN Seq Scan on orders_v1  (cost=0.00..644368.60 rows=64436860 width=16)
        ->  XN Hash  (cost=150000.00..150000.00 rows=15000000 width=21)
              ->  XN Seq Scan on customer_v1 c  (cost=0.00..150000.00 rows=15000000 width=21)
```

6. 이제 아래 쿼리를 2번 실행하고 실행 시간을 기록합니다. 첫 번째 실행은 계획이 컴파일되었는지 확인하는 것입니다. 두 번째가 최종 사용자 경험을 보다 잘 나타냅니다.

```
SELECT c_mktsegment,COUNT(o_orderkey) AS orders_count, sum(l_quantity) as quantity, sum (l_extendedprice) as extendedprice
FROM lineitem
JOIN orders_v1 on l_orderkey = o_orderkey
JOIN customer c on o_custkey = c_custkey
WHERE l_commitdate between '1992-01-01T00:00:00Z' and '1992-12-31T00:00:00Z'
GROUP BY c_mktsegment;
```

종료에 앞서

```
ALTER USER awsuser set enable_result_cache_for_session to true;
```