

**Customer Analytics and Activation**

**Pilot**

**Glue Job Design Document**

**Prepared By: Pradosh Kumar Jena**

**Date:** 07/15/2020

Scope: This is the complete job design which will implement end to end history load for pilot project.

Assumptions:

* The s3 bucket structures are properly laid out
* The source files are in csv format in gzipped compressed format
* The proper roles and permissions are in place to run crawler and glue job.
* The source folder name in s3 bucket and source table name in Data catalog should be of same name

Glue job reference : s\_aadp\_land2fdn\_gj\_order\_caa\_717 ( In sandbox account)

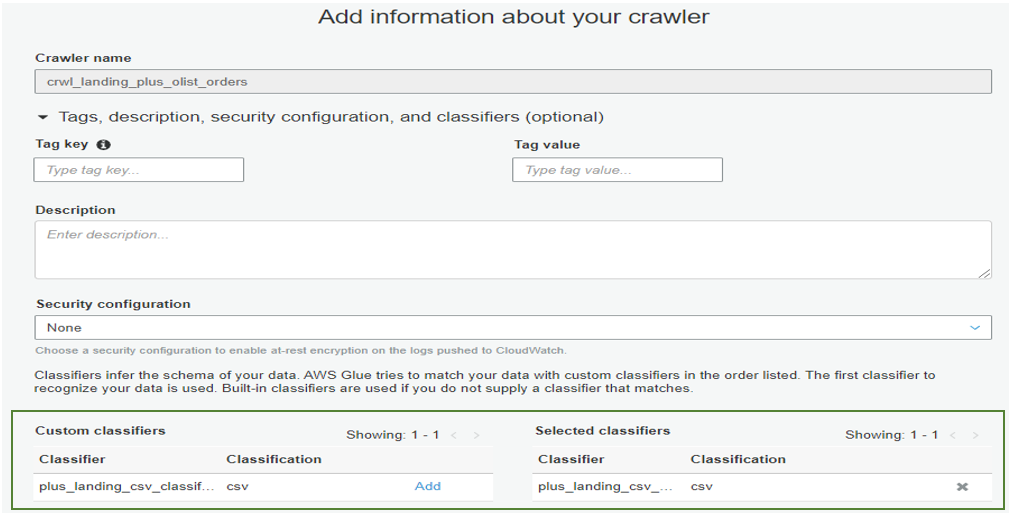
build\_order\_caa\_682 (In datasolutions account)

build\_order\_items\_caa\_682 (In datasolutions account)

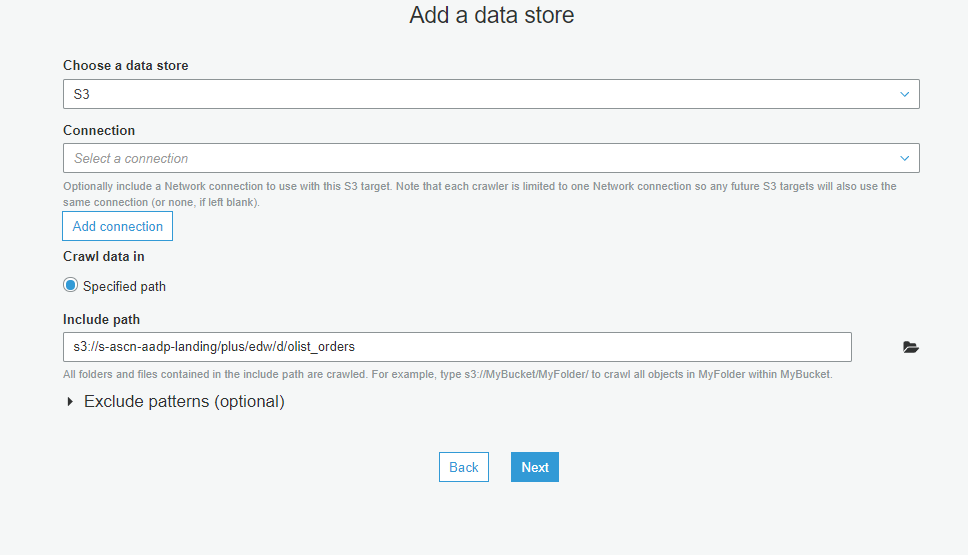
Library file : foundationutil.py

Steps:

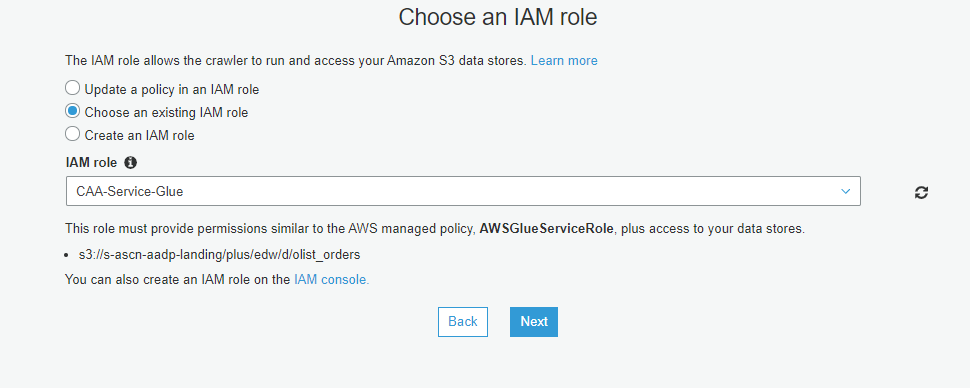
1. The history source data file is available in the landing bucket
2. Run the respective crawler for the source table . Make sure you add the csv classfier.



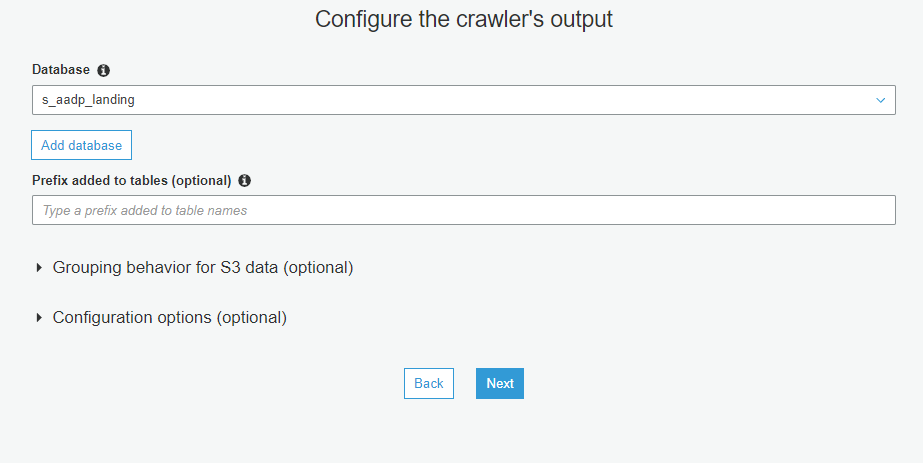
Specify the datastore name



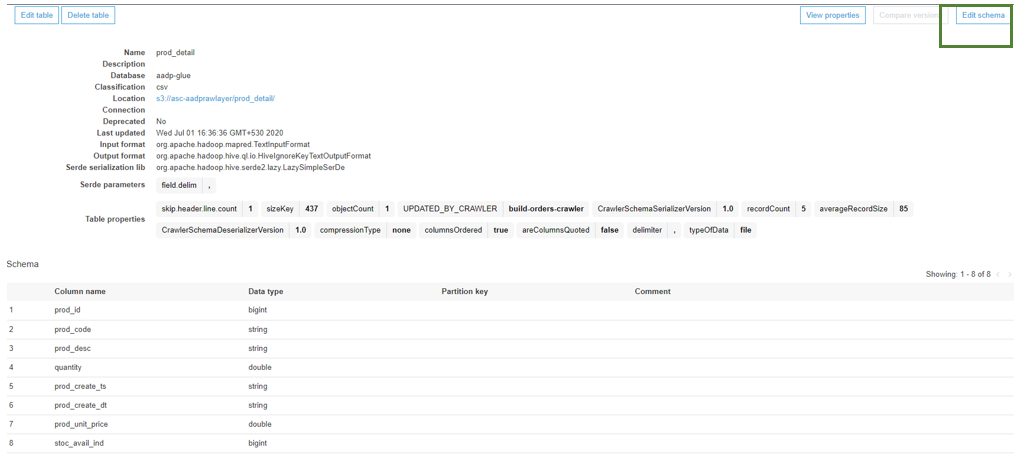
Choose the correct service role

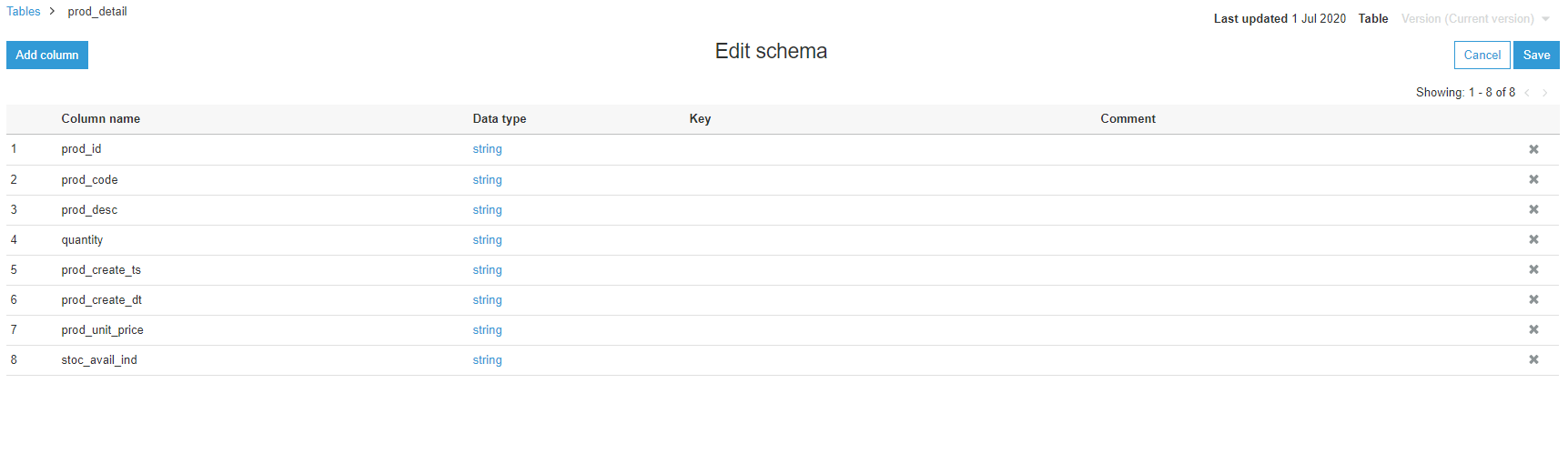


Specify the database name in which you want that table to be created.

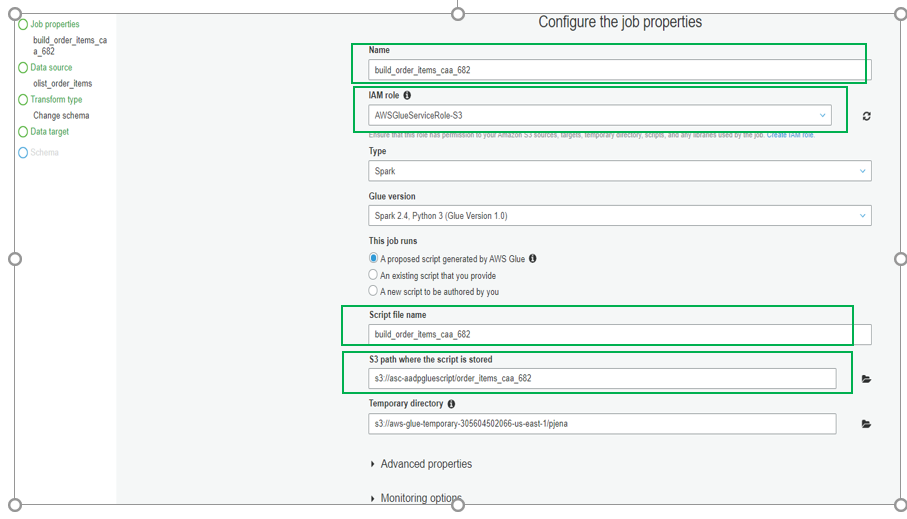


1. Edit the schema if the crawled schema has datatype anything other than string

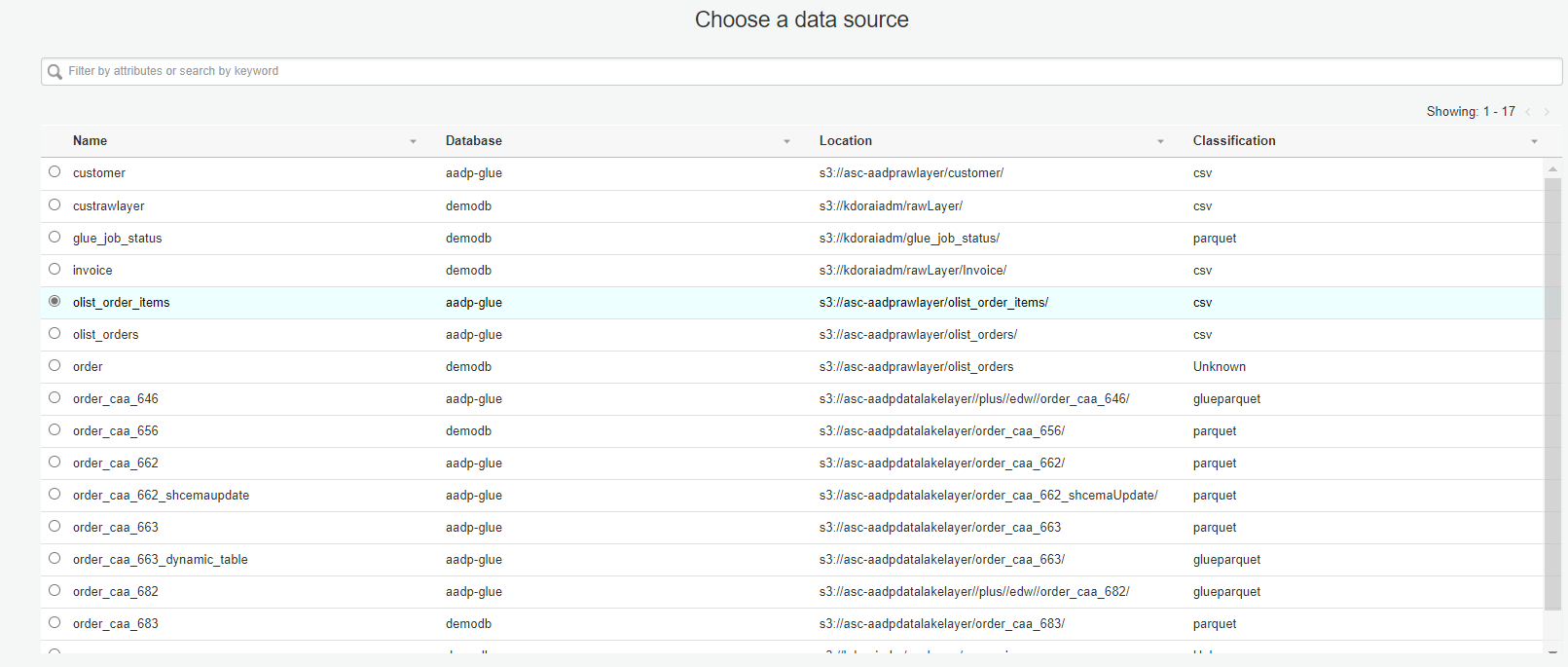


Click edit schema and change all the datatype to string

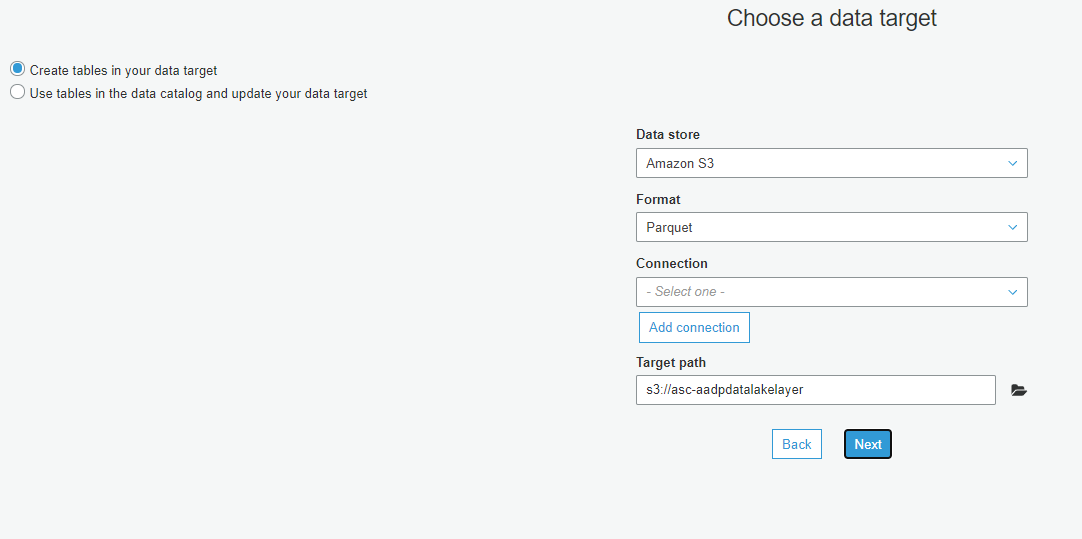
1. Create a Glue job from the console
   * Specify the Job name
   * Specify the appropriate IAM role
   * Check the Script File name is correct or not
   * Specify the Glue script folder path



Choose the correct data store for the source table



Choose the data target, Just specify a bucket path . This is just to generate the Glue pre generated ETL code.



**Details about the Glue job :**

The pre generated glue job will get created with pre-generated ETL code .We are only retaining these sections ( the dynamic frame and apply mapping) for the respective source table.

* This below part of the glue pre generated code we are keeping.

import sys

from awsglue.transforms import \*

from awsglue.utils import getResolvedOptions

from pyspark.context import SparkContext

from awsglue.context import GlueContext

from awsglue.job import Job

## @params: [JOB\_NAME]

args = getResolvedOptions(sys.argv, ['JOB\_NAME'])

sc = SparkContext()

glueContext = GlueContext(sc)

spark = glueContext.spark\_session

job = Job(glueContext)

job.init(args['JOB\_NAME'], args)

datasource0 = glueContext.create\_dynamic\_frame.from\_catalog(database = "aadp-glue", table\_name = "olist\_orders", transformation\_ctx = "datasource0")

applymapping1 = ApplyMapping.apply(frame = datasource0, mappings = [("order\_id", "string", "order\_id", "string"), ("customer\_id", "string", "customer\_id", "string"), ("order\_status", "string", "order\_status", "string"), ("order\_purchase\_timestamp", "string", "order\_purchase\_timestamp", "string"), ("order\_approved\_at", "string", "order\_approved\_at", "string"), ("order\_delivered\_carrier\_date", "string", "order\_delivered\_carrier\_date", "string"), ("order\_delivered\_customer\_date", "string", "order\_delivered\_customer\_date", "string"), ("order\_estimated\_delivery\_date", "string", "order\_estimated\_delivery\_date", "string")], transformation\_ctx = "applymapping1")

* These are the below variables we need to initialize depending upon the job .

# job name

job\_name=args['JOB\_NAME']

# specify source table name and target table name

source\_tbl='olist\_orders'

target\_tbl='order\_caa\_683'

#Specify the data will get load loaded in which folder in target bucket

#source\_prefix='//plus//edw'

source\_prefix=''

# specify primary key and foreign key columns name

map\_keyval = {'olist\_orders\_test\_key':['order\_id'],

'customer\_order\_key':['order\_id','customer\_id'],

'order\_delivered\_key':['order\_id','order\_delivered\_customer\_date']}

#If the table needs to be partitioned , specify the partition key columns

partition\_col="order\_purchase\_timestamp"

#partitionKeys=['chain\_id']

partitionKeys=['transaction\_month']

* We need to call these steps to load data into target.

base\_gdf=create\_base\_frame(applymapping1

,map\_keyval

,glueContext

,spark

,job\_name

,source\_tbl

,audit\_cols=['foundation\_program\_nam','foundation\_ins\_dt'])

target\_gdf=create\_ym\_partitoned\_frame(base\_gdf,partition\_col,glueContext)

write\_to\_sink(target\_gdf,target\_tbl,partitionKeys,glueContext,source\_prefix)

job.commit()

Pseudo code Explanation

#During creation of glue job:

Pass the path of python file in which additional pyspark and python libraries are imported and utility functions are defined .

#Glue pre generated code

Import libraries

Initializing spark and glue context

Use section of the code which reads the glue Data Catalog and creates the source glue data frame

Use section of the code which applies mapping on glue data frame

Removing these pre generated code segments - ResolveChoice , DropNullFields, DataSink

#Custom code begins

#####Write data into sink#####

create\_base\_frame

(

applymapping\_gdf : glue data frame

,map\_keyval : A dictionary where key is name of the key column names and value is list of source columns it's composed of

,glueContext : glue context

,spark : spark session variable

,job\_name : glue job name

,source\_tbl : source table name

,audit\_cols : list of audit columns

)

what it does:

convert dynamic frame to spark data frame

cast the source column dynamically to respective target datatype as per the json schema file - build\_querystring function

create hashed key for primary key and secondary key - hash\_key

add audit columns - add\_auditcol

order the columns

convert back to dynamic frame

what it returns:

ordered glue data frame

#Call this module only if we are creating yearmonth partition column

create\_ym\_partitoned\_frame

(

base\_gdf : ordered glue data frame

,partition\_col : partition column name

,glueContext : glue context)

what it does:

convert dynamic frame to spark data frame

add YYYYMM partition column to data frame - add\_ym\_partition

convert back to dynamic frame

what it returns:

Glue data frame

write\_to\_sink

(target\_gdf : target glue data frame

,target\_tbl : target table name

,partitionKeys : list of partition key column names

,glueContext : glue context

,source\_prefix : subfolders under bucket)

what it does:

create a s3 type sink with enableupdatecatalog true

set the format of the sink to parquet

create the target catalog table

what it returns:

nothing

#Pointers

Improvements : No need to run separate crawler for target

creation of target table on the fly

dynamic query string creation to achieve casting of source datatype to target datatype

Partition logic is decoupled from the base frame building block

If no partition is there , we need to only the base frame function and sink function