Glue Lab:

Important: To avoid namespace collisions, use your initials when naming resources. As this is not an IAM lab and in favour of optimal utilization of time we expect to create or modify IAM roles. Expect the bucket in question (with the dataset) to be in us-east-1.

AWS Glue's dynamic data frames are powerful. They provide a more precise representation of the underlying semi-structured data, especially when dealing with columns or fields with varying types. They also provide powerful primitives to deal with nesting and unnesting.

This example shows how to process CSV files that have unexpected variations in them and convert them into nested and structured Parquet for fast analysis.

Copy the dataset file from the following location to your s3 bucket, like so,

aws s3 cp s3://awsglue-datasets/examples/medicare/Medicare\_Hospital\_Provider.csv

<your s3 bucket/location>/

You may incur some cost for the above operation.

The first step is to crawl this data and put the results into a database called payments in your Data Catalog, as described [here in the Developer Guide](http://docs.aws.amazon.com/glue/latest/dg/console-crawlers.html). Use a GlueServiceRole that provides you with adequate access to read your S3 location. The crawler will read the first 2 MB of data from that file and create one table, medicare, in the payments database in the Data Catalog. If you named your S3 location differently, you will see a different table name. Make a note of the table name before proceeding further.

Use the default VPC with all subnets set up as public. Also create a security group that will allow SSH access from a chosen EC2 instance. Also spawn an EC2 instance in this VPC such that you can start an SSH session to it.

Let’s spin up a Development endpoint as described [here](https://docs.aws.amazon.com/glue/latest/dg/dev-endpoint.html). SSH to Python REPL using the command suggested in the commandline arguments provided in the endpoint details. This will require that the endpoint created is associated with a VPC and Subnet.

Begin by pasting some boilerplate into the DevEndpoint notebook to import the AWS Glue libraries we'll need and set up a single GlueContext.

import sys

from awsglue.transforms import \*

from awsglue.utils import getResolvedOptions

from pyspark.context import SparkContext

from awsglue.context import GlueContext

from awsglue.dynamicframe import DynamicFrame

from awsglue.job import Job

from pyspark.sql import SparkSession

glueContext = GlueContext(SparkContext.getOrCreate())

First, let's see what the schema looks like using Spark DataFrames:

medicare = spark.read.format(

"com.databricks.spark.csv").option(

"header", "true").option(

"inferSchema", "true").load(

's3://<REPLACE WITH YOUR LOCATION>/Medicare\_Hospital\_Provider.csv')

medicare.printSchema()

Review the output from printSchema.

Now, let's see what the schema looks like after we load the data into a DynamicFrame, starting from the metadata that the crawler put in the AWS Glue Data Catalog:

medicare\_dynamicframe = glueContext.create\_dynamic\_frame.from\_catalog(

database = "payments",

table\_name = "medicare")

medicare\_dynamicframe.printSchema()

Again, review the output from printSchema.

The DynamicFrame generated a schema in which provider id could be either a long or a 'string', whereas the DataFrame schema listed Provider Id as being a string. Which one is right? Well, it turns out there are two records (out of 160K records) at the end of the file with strings in that column (these are the erroneous records that we introduced to illustrate our point).

DynamicFrames introduce the notion of a choice type. In this case, it shows that both long and string may appear in that column. The AWS Glue crawler missed the string because it only considered a 2MB prefix of the data. The Spark DataFrame considered the whole dataset, but was forced to assign the most general type to the column (string). In fact, Spark often resorts to the most general case when there are complex types or variations with which it is unfamiliar.

To query the provider id column, we first need to resolve the choice. With DynamicFrames, we can try to convert those string values to long values uaing the resolveChoice transform method with a cast:long option:

medicare\_res = medicare\_dynamicframe.resolveChoice(specs = [('provider id','cast:long')])

medicare\_res.printSchema()

Again, review the output from printSchema.

Where the value was a string that cannot be cast, AWS Glue inserts a null.

Another option is to convert the choice type to a struct, which keeps both types.

Let's take a look at the rows that were anomalous:

medicare\_res.toDF().where("`provider id` is NULL").show()

What do you see?

Let's remove those malformed records now:

medicare\_dataframe = medicare\_res.toDF()

medicare\_dataframe = medicare\_dataframe.where("`provider id` is NOT NULL")

You can always convert a DynamicFrame to and from a Spark DataFrame to take advantage of Spark functionality as well as the special features of DynamicFrames.

Let's turn the payment information into numbers, so analytic engines like Amazon Redshift or Amazon Athena can do their number crunching faster:

from pyspark.sql.functions import udf

from pyspark.sql.types import StringType

chop\_f = udf(lambda x: x[1:], StringType())

medicare\_dataframe = medicare\_dataframe.withColumn(

"ACC", chop\_f(

medicare\_dataframe["average covered charges"])).withColumn(

"ATP", chop\_f(

medicare\_dataframe["average total payments"])).withColumn(

"AMP", chop\_f(

medicare\_dataframe["average medicare payments"]))

medicare\_dataframe.select(['ACC', 'ATP', 'AMP']).show()

What do you see in the output from the show.

These are all still strings in the data. We can use the DynamicFrame's powerful apply\_mapping tranform method to drop, rename, cast, and nest the data so that other data programming langages and sytems can easily access it:

medicare\_tmp\_dyf = DynamicFrame.fromDF(medicare\_dataframe, glueContext, "nested")

medicare\_nest\_dyf = medicare\_tmp\_dyf.apply\_mapping([('drg definition', 'string', 'drg', 'string'),

('provider id', 'long', 'provider.id', 'long'),

('provider name', 'string', 'provider.name', 'string'),

('provider city', 'string', 'provider.city', 'string'),

('provider state', 'string', 'provider.state', 'string'),

('provider zip code', 'long', 'provider.zip', 'long'),

('hospital referral region description', 'string','rr', 'string'),

('ACC', 'string', 'charges.covered', 'double'),

('ATP', 'string', 'charges.total\_pay', 'double'),

('AMP', 'string', 'charges.medicare\_pay', 'double')])

medicare\_nest\_dyf.printSchema()

Have a look at the output from printSchema.

Turning the data back to DataFrame, we can show what it now looks like:

medicare\_nest\_dyf.toDF().show()

Review the output.

Finally, let's write the data out in an optimized Parquet format for Redshift Spectrum or Athena:

glueContext.write\_dynamic\_frame.from\_options(

frame = medicare\_nest\_dyf,

connection\_type = "s3",

connection\_options = {"path": "s3://<SOME OTHER BUCKET>/output-dir/medicare\_parquet"},

format = "parquet")