Joining, Filtering, and Loading Relational Data with AWS Glue

This example shows how to do joins and filters with transforms entirely on DynamicFrames.

1. Crawl our sample dataset

The dataset we'll be using in this example was downloaded from the EveryPolitician (http://everypolitician.org) website into our sample-dataset bucket in S3, at:

```
s3://awsglue-datasets/examples/us-legislators/all.
```

```
$ aws s3 ls --recursive s3://awsglue-datasets/examples/us-legislators/all

2017-08-11 21:30:01     777725 examples/us-legislators/all/areas.json
2017-08-11 21:30:01     378833 examples/us-legislators/all/countries.json
2017-08-11 21:30:01      38985 examples/us-legislators/all/events.json
2017-08-11 21:30:01    2718476 examples/us-legislators/all/memberships.json
2017-08-11 21:29:55     157531 examples/us-legislators/all/organizations.json
2017-08-11 21:29:55    7973806 examples/us-legislators/all/persons.json

# in order to use Athena, each table data must be in its own folder

$ ls -R legislators/
legislators/:
areas  countries  events  memberships  organizations  persons

legislators/areas:
areas.json

legislators/countries:
countries.json

legislators/events:
events.json

legislators/memberships:
memberships.json

legislators/organizations:
organizations.json

legislators/persons:
persons.json

$ aws s3 cp --recursive ./legislators/ s3://wengong-redshift/etl-in/
```

It contains data in JSON format about United States legislators and the seats they have held in the the House of Representatives and the Senate.

For purposes of our example code, we are assuming that you have created an AWS S3 target bucket and folder, which we refer to here as `s3://wengong-redshift/etl-out/us-legislators/` .

The first step is to crawl this data and put the results into a database called `legislators` in your Data Catalog, as described [here in the Developer Guide (http://docs.aws.amazon.com/glue/latest/dg/console-crawlers.html)](http://docs.aws.amazon.com/glue/latest/dg/console-crawlers.html). The crawler will create the following tables in the `legislators` database:

- `persons`
- `memberships`
- `organizations`
- `events`
- `areas`
- `countries_r`

This is a semi-normalized collection of tables containing legislators and their histories.

```
sql> show create table events
```

2. Getting started

We will write a script that:

1. Combines persons, organizations, and membership histories into a single legislator history data set. This is often referred to as de-normalization.
2. Separates out the senators from the representatives.
3. Writes each of these out to separate parquet files for later analysis.

Begin by running some boilerplate to import the AWS Glue libraries we'll need and set up a single `GlueContext` .

In [1]:
```
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

glueContext = GlueContext(SparkContext.getOrCreate())
```

```
Starting Spark application
```

ID	YARN Application ID	Kind	State	Spark UI	
2	application_1599497572147_0003	pyspark	idle	Link (http://ip-172-32-179-184.ec2.internal:20888/proxy/application_1599497572147_0003/)	165.ec2.internal:8042/node/c

```
SparkSession available as 'spark'.
```

In [2]:
```
S3_OUT_DIR = "s3://wengong-redshift/etl-out/us-legislators"
```

3. Checking the schemas that the crawler identified

Next, you can easily examine the schemas that the crawler recorded in the Data Catalog. For example, to see the schema of the `persons` table, run the following code:

In [3]:
```
persons = glueContext.create_dynamic_frame.from_catalog(database="legislators", table_name="persons"
print("Count: " + str(persons.count()))
persons.printSchema()
```

```
 |    |-- element: struct
 |    |    |-- scheme: string
 |    |    |-- identifier: string
 |-- other_names: array
 |    |-- element: struct
 |    |    |-- lang: string
 |    |    |-- note: string
 |    |    |-- name: string
 |-- sort_name: string
 |-- images: array
 |    |-- element: struct
 |    |    |-- url: string
 |-- given_name: string
 |-- birth_date: string
 |-- id: string
 |-- contact_details: array
 |    |-- element: struct
 |    |    |-- type: string
 |    |    |-- value: string
 |-- death_date: string
```

Each person in the table is a member of some congressional body.

Look at the schema of the `memberships` table:

In [4]:
```
memberships = glueContext.create_dynamic_frame.from_catalog(database="legislators", table_name="memb
print("Count: " + str(memberships.count()))
memberships.printSchema()
```

```
Count: 10439
root
|-- area_id: string
|-- on_behalf_of_id: string
|-- organization_id: string
|-- role: string
|-- person_id: string
|-- legislative_period_id: string
|-- start_date: string
|-- end_date: string
```

Organizations are parties and the two chambers of congress, the Senate and House. Look at the schema of the `organizations` table:

In [5]:
```
orgs = glueContext.create_dynamic_frame.from_catalog(database="legislators", table_name="organizatic
print("Count: " + str(orgs.count()))
orgs.printSchema()
```

```
Count: 13
root
|-- identifiers: array
|    |-- element: struct
|    |    |-- scheme: string
|    |    |-- identifier: string
|-- other_names: array
|    |-- element: struct
|    |    |-- lang: string
|    |    |-- note: string
|    |    |-- name: string
|-- id: string
|-- classification: string
|-- name: string
|-- links: array
|    |-- element: struct
|    |    |-- note: string
|    |    |-- url: string
|-- image: string
|-- seats: int
|-- type: string
```

4. Filtering

Let's only keep the fields that we want and rename `id` to `org_id`. The dataset is small enough that we can look at the whole thing. The `toDF()` converts a DynamicFrame to a Spark DataFrame, so we can apply the transforms that already exist in SparkSQL:

In [6]:
```
orgs = orgs.drop_fields(['other_names','identifiers']).rename_field('id', 'org_id').rename_field('na
orgs.toDF().show()
```

```
+--------------+--------------------+--------------------+--------------------+--------------------
+-----+-----------+
|classification|              org_id|            org_name|               links|               image
|seats|       type|
+--------------+--------------------+--------------------+--------------------+--------------------
+-----+-----------+
|         party|            party/al|                  AL|                null|                null
| null|       null|
|         party|      party/democrat|            Democrat|[[website, http:/...|https://upload.wi...
| (http:/...|https://upload.wi...|) null|       null|
|         party|party/democrat-li...|    Democrat-Liberal|[[website, http:/...| (http:/...|)
null| null|       null|
|   legislature|d56acebe-8fdc-47b...|House of Represen...|                null|                null
|  435|lower house|
|         party|   party/independent|         Independent|                null|                null
| null|       null|
|         party|party/new_progres...|     New Progressive|[[website, http:/...|https://upload.wi...
| (http:/...|https://upload.wi...|) null|       null|
|         party|party/popular_dem...|    Popular Democrat|[[website, http:/...| (http:/...|)
null| null|       null|
|         party|    party/republican|          Republican|[[website, http:/...|https://upload.wi...
| (http:/...|https://upload.wi...|) null|       null|
|         party|party/republican-...|Republican-Conser...|[[website, http:/...| (http:/...|)
null| null|       null|
|         party|      party/democrat|            Democrat|[[website, http:/...|https://upload.wi...
| (http:/...|https://upload.wi...|) null|       null|
|         party|   party/independent|         Independent|                null|                null
| null|       null|
|         party|    party/republican|          Republican|[[website, http:/...|https://upload.wi...
| (http:/...|https://upload.wi...|) null|       null|
|   legislature|8fa6c3d2-71dc-478...|              Senate|                null|                null
|  100|upper house|
+--------------+--------------------+--------------------+--------------------+--------------------
+-----+-----------+
```

Let's look at the `organizations` that appear in `memberships` :

In [7]:
```
memberships.select_fields(['organization_id']).toDF().distinct().show()
```

```
+--------------------+
|     organization_id|
+--------------------+
|d56acebe-8fdc-47b...|
|8fa6c3d2-71dc-478...|
+--------------------+
```

5. Putting it together

Now let's join these relational tables to create one full history table of legislator memberships and their correponding organizations, using AWS Glue.

- First, we join `persons` and `memberships` on `id` and `person_id` .
- Next, join the result with orgs on `org_id` and `organization_id` .
- Then, drop the redundant fields, `person_id` and `org_id` .

We can do all these operations in one (extended) line of code:

In [8]:
```
l_history = Join.apply(orgs,
                       Join.apply(persons, memberships, 'id', 'person_id'),
                       'org_id', 'organization_id').drop_fields(['person_id', 'org_id'])
print("Count: " + str(l_history.count()))
l_history.printSchema()
```

```
Count: 10439
root
|-- role: string
|-- seats: int
|-- org_name: string
|-- links: array
|    |-- element: struct
|    |    |-- note: string
|    |    |-- url: string
|-- type: string
|-- sort_name: string
|-- area_id: string
|-- images: array
|    |-- element: struct
|    |    |-- url: string
|-- on_behalf_of_id: string
|-- other_names: array
|    |-- element: struct
|    |    |-- lang: string
|    |    |-- note: string
|    |    |-- name: string
|-- name: string
|-- birth_date: string
|-- organization_id: string
|-- gender: string
|-- classification: string
|-- death_date: string
|-- legislative_period_id: string
|-- identifiers: array
|    |-- element: struct
|    |    |-- scheme: string
|    |    |-- identifier: string
|-- image: string
|-- given_name: string
|-- start_date: string
|-- family_name: string
|-- id: string
```

```
|-- contact_details: array
|    |-- element: struct
|    |    |-- type: string
|    |    |-- value: string
|-- end_date: string
```

Great! We now have the final table that we'd like to use for analysis. Let's write it out in a compact, efficient format for analytics, i.e. Parquet, that we can run SQL over in AWS Glue, Athena, or Redshift Spectrum.

The following call writes the table across multiple files to support fast parallel reads when doing analysis later:

In [9]:
```
glueContext.write_dynamic_frame.from_options(frame = l_history,
              connection_type = "s3",
              connection_options = {"path": f"{S3_OUT_DIR}/legislator_history"},
              format = "parquet")
```

```
<awsglue.dynamicframe.DynamicFrame object at 0x7f7b1eae8c18>
```

In [10]:
```
l_history.toDF().show(5, False)
```

```
ilingual, Fred Thompson], [it, multilingual, Fred Thompson], [ja, multilingual, フレッド・トンプソ
ン], [ko, multilingual, 프레드 톰프슨], [lb, multilingual, Fred Thompson], [nb, multilingual, Fred T
hompson], [nds, multilingual, Fred Thompson], [nl, multilingual, Fred Thompson], [nn, multilingua
l, Fred Thompson], [pl, multilingual, Fred Thompson], [pt, multilingual, Fred Thompson], [ru, mul
tilingual, Томпсон, Фред Далтон], [sh, multilingual, Fred Thompson], [sv, multilingual, Fred Thom
pson], [tr, multilingual, Fred Thompson], [uk, multilingual, Фред Томпсон], [yi, multilingual, פר
עד טאמפסאן], [zh, multilingual, 弗雷德·汤普森], [zh-cn, multilingual, 弗雷德·汤普森], [zh-hans, multil
ingual, 弗雷德·汤普森], [zh-hant, multilingual, 弗雷德·湯普森], [zh-hk, multilingual, 弗雷德·湯普森],
 [zh-sg, multilingual, 弗雷德·汤普森], [zh-tw, multilingual, 弗雷德·湯普森]]|Fred Thompson|1942-08-19
|8fa6c3d2-71dc-4788-b9f8-4ca70d5a7d85|male  |legislature   |2015-11-22|term/103            |[[al
lmovie, p70694], [allocine, 104570], [bioguide, T000457], [bnf, 14232976w], [csfd, 38738], [cspa
n, fredthompson], [dnf, 216827], [elonet, 240769], [everypolitician_legacy, T000457], [fast, 1851
010], [filmportal_de, 05e977627377476ab768ff9f16807faa], [freebase, /m/02p8v8], [gnd, 106127705
4], [google_entity_id, kg:/m/02p8v8], [govtrack, 300158], [icpsr, 49503], [imdb, nm0000669], [isn
i, 0000 0000 8750 2388], [kinopoisk, 103919], [lcauth, n2001021403], [lis, S237], [munzinger, 000
00026234], [nndb, 413/000024341], [opensecrets, N00003136], [politifact, fred-thompson], [port, 2
1055], [quora, Fred-Thompson-41], [scope, 19139], [sfdb, 112414], [snac, w6kd9vbg], [sudoc, 03365
3399], [thomas, 01447], [uscongress, T000457], [viaf, 294402294], [wikidata, Q298016], [wikipedi
a, Fred Thompson], [wikitree, Thompson-25193]]|https://theunitedstates.io/images/congress/origina
l/T000457.jpg|Fred      |1994-12-02|Thompson  |0fcf1022-8066-42f7-86ec-d9d450f69a0e|null
```

To put all the history data into a single file, we need to convert it to a data frame, repartition it, and write it out.

In [11]:
```
s_history = l_history.toDF().repartition(1)
s_history.write.parquet(f"{S3_OUT_DIR}/legislator_single")
```

```
An error was encountered:
'path s3://wengong-redshift/etl-out/us-legislators/legislator_single already exists.;'
Traceback (most recent call last):
  File "/mnt/yarn/usercache/livy/appcache/application_1599497572147_0003/container_1599497572147_0003_01_000001/pyspark.zip/pyspark/sql/readwriter.py", line 839, in parquet
    self._jwrite.parquet(path)
  File "/mnt/yarn/usercache/livy/appcache/application_1599497572147_0003/container_1599497572147_0003_01_000001/py4j-0.10.7-src.zip/py4j/java_gateway.py", line 1257, in __call__
    answer, self.gateway_client, self.target_id, self.name)
  File "/mnt/yarn/usercache/livy/appcache/application_1599497572147_0003/container_1599497572147_0003_01_000001/pyspark.zip/pyspark/sql/utils.py", line 69, in deco
    raise AnalysisException(s.split(': ', 1)[1], stackTrace)
pyspark.sql.utils.AnalysisException: 'path s3://wengong-redshift/etl-out/us-legislators/legislator_single already exists.;'
```

Or if you want to separate it by the Senate and the House:

In [12]:
```
l_history.toDF().write.parquet(f"{S3_OUT_DIR}/legislator_part", partitionBy=['org_name'])
```

```
An error was encountered:
'path s3://wengong-redshift/etl-out/us-legislators/legislator_part already exists.;'
Traceback (most recent call last):
  File "/mnt/yarn/usercache/livy/appcache/application_1599497572147_0003/container_1599497572147_0003_01_000001/pyspark.zip/pyspark/sql/readwriter.py", line 839, in parquet
    self._jwrite.parquet(path)
  File "/mnt/yarn/usercache/livy/appcache/application_1599497572147_0003/container_1599497572147_0003_01_000001/py4j-0.10.7-src.zip/py4j/java_gateway.py", line 1257, in __call__
    answer, self.gateway_client, self.target_id, self.name)
  File "/mnt/yarn/usercache/livy/appcache/application_1599497572147_0003/container_1599497572147_0003_01_000001/pyspark.zip/pyspark/sql/utils.py", line 69, in deco
    raise AnalysisException(s.split(': ', 1)[1], stackTrace)
pyspark.sql.utils.AnalysisException: 'path s3://wengong-redshift/etl-out/us-legislators/legislator_part already exists.;'
```

6. Writing to Relational Databases

AWS Glue makes it easy to write it to relational databases like Redshift even with semi-structured data. It offers a transform, `relationalize()` , that flattens DynamicFrames no matter how complex the objects in the frame may be.

Using the `l_history` DynamicFrame in our example, we pass in the name of a root table (`hist_root`) and a temporary working path to `relationalize` , which returns a `DynamicFrameCollection` . We then list the names of the DynamicFrames in that collection:

In [13]:
```
dfc = l_history.relationalize("hist_root", f"{S3_OUT_DIR}/temp/")
dfc.keys()
```

```
dict_keys(['hist_root', 'hist_root_links', 'hist_root_images', 'hist_root_identifiers', 'hist_root_
other_names', 'hist_root_contact_details'])
```

Relationalize broke the history table out into 6 new tables: a root table containing a record for each object in the dynamic frame, and auxiliary tables for the arrays. Array handling in relational databases is often sub-optimal, especially as those arrays become large. Separating out the arrays into separate tables makes the queries go much faster.

Let's take a look at the separation by examining `contact_details` :

In [14]:
```
l_history.select_fields('contact_details').printSchema()
dfc.select('hist_root_contact_details').toDF().where("id = 10 or id = 75").orderBy(['id','index']).s
```

```
root
|-- contact_details: array
|    |-- element: struct
|    |    |-- type: string
|    |    |-- value: string

+---+-----+----------------------+-----------------------+
| id|index|contact_details.val.type|contact_details.val.value|
+---+-----+----------------------+-----------------------+
| 10|    0|                   fax|           202-228-3027|
| 10|    1|                 phone|           202-224-6542|
| 10|    2|               twitter|             SenSchumer|
| 75|    0|                   fax|           202-224-6747|
| 75|    1|                 phone|           202-224-3934|
+---+-----+----------------------+-----------------------+
```

The `contact_details` field was an array of structs in the original DynamicFrame. Each element of those arrays is a separate row in the auxiliary table, indexed by `index`. The `id` here is a foreign key into the `hist_root` table with the key `contact_details`.

In [15]:
```
dfc.select('hist_root').toDF().where("contact_details = 10 or contact_details = 75").select(['id', '
```

```
+--------------------+----------+-----------+---------------+
|                  id|given_name|family_name|contact_details|
+--------------------+----------+-----------+---------------+
|60ae8ebc-b581-44e...|   Charles|    Schumer|             10|
|0d69087e-f098-460...|    Daniel|     Inouye|             75|
+--------------------+----------+-----------+---------------+
```

Notice in the commands above that we used `toDF()` and subsequently a `where` expression to filter for the rows that we wanted to see.

So, joining the `hist_root` table with the auxiliary tables allows you to:

- Load data into databases without array support.
- Query each individual item in an array using SQL.

We already have a connection set up called `redshift3`. To create your own, see [this topic in the Developer Guide (http://docs.aws.amazon.com/glue/latest/dg/populate-add-connection.html)](http://docs.aws.amazon.com/glue/latest/dg/populate-add-connection.html). Let's write this collection into Redshift by cycling through the DynamicFrames one at a time:

In [*]:
```
for df_name in dfc.keys():
        m_df = dfc.select(df_name)
        print("Writing to Redshift table: " + df_name)
        glueContext.write_dynamic_frame.from_jdbc_conf(frame = m_df,
                                                       catalog_connection = "redshiftc1db",
                                                       connection_options = {"dbtable": df_name, "da
                                                       redshift_tmp_dir = f"{S3_OUT_DIR}/temp/")
```

Progress:

Notice in the commands above that we used `toDF()` and subsequently a `where` expression to filter for the rows that we wanted to see.

So, joining the `hist_root` table with the auxiliary tables allows you to:

- Load data into databases without array support.
- Query each individual item in an array using SQL.

We already have a connection set up called `redshift3`. To create your own, see [this topic in the Developer Guide (http://docs.aws.amazon.com/glue/latest/dg/populate-add-connection.html)](http://docs.aws.amazon.com/glue/latest/dg/populate-add-connection.html). Let's write this collection into Redshift by cycling through the DynamicFrames one at a time:

```
for df_name in dfc.keys():
    m_df = dfc.select(df_name)
    print("Writing to Redshift table: " + df_name)
    glueContext.write_dynamic_frame.from_jdbc_conf(frame = m_df,
                                                   catalog_connection = "redshift3",
                                                   connection_options = {"dbtable": df_name,
"database": "testdb"},
                                                   redshift_tmp_dir = "s3://glue-sample-targ
et/temp-dir/")
```

Here's what the tables look like in Redshift. (We connected to Redshift through psql.)

```
testdb=# \d
                   List of relations
 schema |           name           | type  |   owner
--------+--------------------------+-------+-----------
 public | hist_root                | table | test_user
 public | hist_root_contact_details | table | test_user
 public | hist_root_identifiers     | table | test_user
 public | hist_root_images          | table | test_user
 public | hist_root_links           | table | test_user
 public | hist_root_other_names     | table | test_user
(6 rows)

testdb=# \d hist_root_contact_details
             Table "public.hist_root_contact_details"
          Column           |           Type           | Modifiers
---------------------------+--------------------------+-----------
 id                        | bigint                   |
 index                     | integer                  |
 contact_details.val.type  | character varying(65535) |
 contact_details.val.value | character varying(65535) |

testdb=# \d hist_root
                   Table "public.hist_root"
        Column         |           Type           | Modifiers
-----------------------+--------------------------+-----------
 role                  | character varying(65535) |
 seats                 | integer                  |
 org_name              | character varying(65535) |
 links                 | bigint                   |
 type                  | character varying(65535) |
 sort_name             | character varying(65535) |
 area_id               | character varying(65535) |
 images                | bigint                   |
 on_behalf_of_id       | character varying(65535) |
 other_names           | bigint                   |
 birth_date            | character varying(65535) |
 name                  | character varying(65535) |
```

```
 name                  | character varying(65535) |
 organization_id       | character varying(65535) |
 gender                | character varying(65535) |
 classification        | character varying(65535) |
 legislative_period_id | character varying(65535) |
 identifiers           | bigint                   |
 given_name            | character varying(65535) |
 image                 | character varying(65535) |
 family_name           | character varying(65535) |
 id                    | character varying(65535) |
 death_date            | character varying(65535) |
 start_date            | character varying(65535) |
 contact_details       | bigint                   |
 end_date              | character varying(65535) |
```

Now you can query these tables using SQL in Redshift:

```
testdb=# select * from hist_root_contact_details where id = 10 or id = 75 order by id, inde
x;
```

With this result:

```
 id | index | contact_details.val.type | contact_details.val.value
----+-------+--------------------------+---------------------------
 10 |     0 | fax                      |
 10 |     1 |                          | 202-225-1314
 10 |     2 | phone                    |
 10 |     3 |                          | 202-225-3772
 10 |     4 | twitter                  |
 10 |     5 |                          | MikeRossUpdates
 75 |     0 | fax                      |
 75 |     1 |                          | 202-225-7856
 75 |     2 | phone                    |
 75 |     3 |                          | 202-225-2711
 75 |     4 | twitter                  |
 75 |     5 |                          | SenCapito
(12 rows)
```

Conclusion

Overall, AWS Glue is quite flexible allowing you to do in a few lines of code, what normally would take days to write. The entire source to target ETL scripts from end-to-end can be found in the accompanying Python file, join_and_relationalize.py (join_and_relationalize.py).

In [ ]:

```

```