Going Serverless - an Introduction to AWS Glue

Michael Rainey | Oaktable World 2018
what is serverless
what is

serverless ==

function as a service
what is serverless == fully managed
what is serverless == allowing developers to focus on ...development
“…any platform that provides utility while completely abstracting scaling and reliability”

Nick Rockwell, NYTimes CTO
Where serverless fits

Source: http://gunnarpeipman.com/2018/03/serverless-architecture/
In the beginning...

- AWS Lambda in 2014
  - Function as a Service (FaaS)
- Others
  - Google Cloud Functions, Microsoft Azure Functions, The Fn Project (Oracle)
  - Amazon Athena, Google BigQuery, Snowflake, etc.

- Why serverless?
  - Pay as you go
  - Zero administration - “fully managed”
    - You’re not provisioning infrastructure, but you’re still in charge of the compute
  - Scale automatically
  - Ship code faster

- Serverless use cases
  - Websites
  - Post processing updates
  - Data ingestion pipelines
  - Backups/log analysis/etc
  - Capture customer social media data to store in your CRM
Serverless: challenges / limitations

- Monitoring and debugging
- Integration testing can be difficult
- Maintaining state of application across multiple functions is tricky
- Startup latency
- Built for small, short processes (FaaS typically <5 min)
- Vendor lock-in
  - Access to multi-cloud data can be difficult
AWS Glue

Serverless ETL
AWS Glue

“…a fully managed extract, transform, and load (ETL) service…”

• Components:
  • Data Catalog
  • Crawlers
  • ETL jobs/scripts
  • Job scheduler

• Useful for…
  • …running serverless queries against S3 buckets and relational data
  • …creating event-driven ETL pipelines
  • …automatically discovering and cataloging your enterprise data assets
AWS Glue architecture

Source: https://docs.aws.amazon.com/athena/latest/ug/glue-athena.html
Data catalog

“…a central repository to store structural and operational metadata for all your data assets.”

- Apache Hive Metastore compatible
  - Can act as metadata repository for EMR, Athena, Redshift Spectrum
- Tables
  - Organized in databases
  - Structure and metadata
  - Add using the wizard or run a crawler
- Data sources:
  - Amazon S3, Oracle, PostgreSQL, Amazon Redshift, Amazon Aurora, MySQL, MariaDB, MS SQL Server, custom JDBC

<table>
<thead>
<tr>
<th>Name</th>
<th>Database</th>
<th>Location</th>
<th>Classification</th>
</tr>
</thead>
<tbody>
<tr>
<td>10_45_0.23</td>
<td>int12102_sh</td>
<td>s3://gluent.backup/user/gluent/backup/10.45.0.23/</td>
<td>orc</td>
</tr>
<tr>
<td>10_45_1.88</td>
<td>int12102_sh</td>
<td>s3://gluent.backup/user/gluent/backup/10.45.1.88/</td>
<td>parquet</td>
</tr>
<tr>
<td>channels</td>
<td>int12102_sh</td>
<td>s3://gluent.backup/user/gluent/backup/sh.db/ch...</td>
<td>orc</td>
</tr>
<tr>
<td>channels_4179ce4c147...</td>
<td>int12102_sh</td>
<td>s3://gluent.backup/user/gluent/backup/sh_test.db/c...</td>
<td>parquet</td>
</tr>
<tr>
<td>countries</td>
<td>int12102_sh</td>
<td>s3://gluent.backup/user/gluent/backup/internal_de...</td>
<td>parquet</td>
</tr>
<tr>
<td>countries_8541194d2ae...</td>
<td>int12102_sh</td>
<td>s3://gluent.backup/user/gluent/backup/sh_test.db/c...</td>
<td>parquet</td>
</tr>
<tr>
<td>customers</td>
<td>int12102_sh</td>
<td>s3://gluent.backup/user/gluent/backup/internal_de...</td>
<td>parquet</td>
</tr>
</tbody>
</table>
Crawlers

“...connects to one or more data stores, determines the data structures, and writes tables into the Data Catalog.”

- Include/exclude specific objects
- Detect Hive style partitions
- Detect schema changes as structures evolve

- Classifiers - automatic schema inference
  - Detects format of the data to generate the correct schema
  - Built-in and custom (written in Grok, JSON, or XML)
  - Can invoke list of classifiers

Cast a `num` field to an `int` data type

```bash
%%NUMBER:num:int
```
Jobs

“A job is the business logic that performs the extract, transform, and load (ETL) work in AWS Glue.”

• PySpark or Scala scripts, generated by AWS Glue
  • Use Glue generated scripts or provide your own
• Built-in transforms to process data
  • The data structure used, called a DynamicFrame, is an extension to an Apache Spark SQL DataFrame
• Visual dataflow can be generated
• Development endpoint available to write scripts in a notebook or IDE
Jobs - built-in transforms

- **ApplyMapping**
  - Maps source and target columns

- **Filter**
  - Load new DynamicFrame based on filtered records

- **Join**
  - Joins two DynamicFrames

- **Map**
  - Applies a function to the records in a DynamicFrame

- **SelectFields**
  - Output selected fields to new DynamicFrame

- **Spigot**
  - Sampling of data written to S3

- **SplitFields**
  - Split fields into two new DynamicFrames

- **SplitRows**
  - Split rows into two new DynamicFrames based on a predicate

```python
# Join the frames to create history
l_history = Join.apply(persons, memberships, 'id', 'person_id')
```
Running a job in AWS Glue

“With AWS Glue, you only pay for the time your ETL job takes to run.”

- Fire off the ETL using the job scheduler, events, or manually invoke
- Data processing units (DPUs) used to calculate processing capacity & cost
  - A single DPU = 4 vCPUs compute and 16 GB of memory
  - Can be a custom set value from 2 - 100
  - Billed $0.44 per DPU-Hour in increments of 1 second
  - 10-minute minimum duration for each job

ETL job example:
Consider an ETL job that runs for 10 minutes and consumes 6 DPUs. Since your job ran for 1/6th of an hour and consumed 6 DPUs, you will be billed 6 DPUs * 1/6 hour at $0.44 per DPU-Hour or a total of $0.44.
Creating a job via the wizard
Creating a job via the wizard

Map the source columns to target columns.

Verify the mappings created by AWS Glue. Change mappings by choosing other columns with **Map to target**. You can **Clear** all mappings and **Reset** to default AWS Glue mappings. AWS Glue generates your script with the defined mappings.

**Source**

<table>
<thead>
<tr>
<th>Column name</th>
<th>Data type</th>
<th>Map to target</th>
</tr>
</thead>
<tbody>
<tr>
<td>cust_id</td>
<td>bigint</td>
<td></td>
</tr>
<tr>
<td>cust_first_nar</td>
<td>string</td>
<td></td>
</tr>
<tr>
<td>cust_last_nar</td>
<td>string</td>
<td></td>
</tr>
<tr>
<td>cust_gender</td>
<td>string</td>
<td></td>
</tr>
<tr>
<td>cust_year_of_j</td>
<td>bigint</td>
<td>avg_by_occupation</td>
</tr>
<tr>
<td>cust_marital_s</td>
<td>string</td>
<td></td>
</tr>
<tr>
<td>cust_street_ar</td>
<td>string</td>
<td></td>
</tr>
<tr>
<td>cust_postal_c</td>
<td>string</td>
<td></td>
</tr>
<tr>
<td>cust_city</td>
<td>string</td>
<td>occupation</td>
</tr>
<tr>
<td>cust_city_id</td>
<td>bigint</td>
<td></td>
</tr>
<tr>
<td>cust_state_pr</td>
<td>string</td>
<td></td>
</tr>
<tr>
<td>cust_state_pr bigint</td>
<td></td>
<td></td>
</tr>
<tr>
<td>country_id</td>
<td>bigint</td>
<td>amount_sold</td>
</tr>
</tbody>
</table>

**Target**

<table>
<thead>
<tr>
<th>Column name</th>
<th>Data type</th>
</tr>
</thead>
<tbody>
<tr>
<td>avg_by_occupation</td>
<td>decimal(1...)</td>
</tr>
<tr>
<td>occupation</td>
<td>string</td>
</tr>
<tr>
<td>amount_sold</td>
<td>decimal(6,4)</td>
</tr>
<tr>
<td>avg_by_education</td>
<td>decimal(1...)</td>
</tr>
<tr>
<td>cust_name</td>
<td>string</td>
</tr>
<tr>
<td>education</td>
<td>string</td>
</tr>
<tr>
<td>avg_total</td>
<td>decimal(1...)</td>
</tr>
</tbody>
</table>
Creating a job via the wizard
Job script editing

cust_demo_sales_t = cust_demo_sales.toDF()

cust_demo_sales_sql = spark.sql("SELECT concat(cust_last_name, ',', cust_first_name) cust_name, education, occupation, amount_sold,
    avg(amount_sold) OVER (partition by education) avg_by_education,
    avg(amount_sold) OVER (partition by occupation) avg_by_occupation,
    avg(amount_sold) OVER () avg_total
FROM cust_demo_sales_tbl
WHERE cast(cust_city as varchar) = 'Coventry'")

cust_demo_sales_final = DynamicFr.fromDF(cust_demo_sales_sql, glueContext, "cust_demo_sales_final")
AWS Glue in action
Connecting to Snowflake
Setup AWS Glue and Snowflake

- Drop the Snowflake Spark Connector and Snowflake JDBC Driver JAR files somewhere Glue can access them – like an S3 bucket
- Add path to “dependent jars” in job properties
## Setup

- Add parameters for connection to Snowflake

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>--SCHEMA</td>
<td>PUBLIC</td>
</tr>
<tr>
<td>--URL</td>
<td>sfcsupport.snowflakecompu</td>
</tr>
<tr>
<td>--WAREHOUSE</td>
<td>MRAINEY_WH</td>
</tr>
<tr>
<td>--ACCOUNT</td>
<td>sfcsupport</td>
</tr>
<tr>
<td>--USERNAME</td>
<td>MRAINEY</td>
</tr>
<tr>
<td>--DB</td>
<td>MRAINEY_TEST</td>
</tr>
<tr>
<td>--PASSWORD</td>
<td>myCoolPassword</td>
</tr>
</tbody>
</table>
Connecting to Snowflake

```python
import sys
import getResolvedOptions
import SparkContext
import GlueContext
import Job
import Job
import java_import
SNOWFLAKE_SOURCE_NAME = "net.snowflake.spark.snowflake"

## @params: [JOB_NAME, URL, ACCOUNT, WAREHOUSE, DB, SCHEMA, USERNAME, PASSWROD]
args = getResolvedOptions(sys.argv, ['JOB_NAME', 'URL', 'ACCOUNT', 'WAREHOUSE', 'DB', 'SCHEMA', 'USERNAME', 'PASSWORD'])
sc = SparkContext()
spark = GlueContext(sc).
job = Job(spark)
job.init(args['JOB_NAME'], args)
java_import(spark._jvm, "net.snowflake.spark.snowflake")

## enable query pushdown to Snowflake
spark._jvm.net.snowflake.spark.snowflake.SnowflakeConnectorUtils.enablePushdownSession(
    spark._jvm.org.apache.spark.sql.SparkSession.builder().getOrCreate())

sfOptions = {
    "sfURL" : args['URL'],
    "sfAccount" : args['ACCOUNT'],
    "sfUser" : args['USERNAME'],
    "sfPassword" : args['PASSWORD'],
    "sfDatabase" : args['DB'],
    "sfSchema" : args['SCHEMA'],
    "sfWarehouse" : args['WAREHOUSE'],
}
```
Running a Glue job

```java
## Read from S3 file - employee.csv - into a Spark Data Frame
dfEmployee = glueContext.create_dynamic_frame \
    .from_catalog(database = "oggdata", table_name = "employee").toDF()

## Read from S3 file - montlyossilane.csv - into a Spark Data Frame
dfMont = glueContext.create_dynamic_frame \
    .from_catalog(database = "oggdata", table_name = "monthlyossilane").toDF()

## Aggregate data

dfGroup = dfMont.groupBy("person_id", "month")
    .agg(sum("amount").alias("sum(AMOUNT)"))
    .groupBy("person_id")
    .show()

## Join aggregated data with employee data

dfEmp = dfEmployee.join(dfGroup, "person_id", "left")

## Select relevant columns

dfEmployee = dfEmp.select("FIRST_NAME", "LAST_NAME", "MONTH", "sum(AMOUNT)"")

## Write results to S3

dfEmployee.write.option("**sfOptions") \
    .option("dbtable", "MRAINEY_TEST.PUBLIC.EMPLOYEE_ANNUAL_SALES").mode("overwrite").save()
job.commit()
```
Why use AWS Glue with Snowflake?

- Event Driven ETL Pipelines
  - Process and land data in S3
  - Trigger script to move data to Snowflake (COPY)
- ETL – S3 as a source, Snowflake as target
  - Process using Spark within Glue
- ETL – Snowflake as a source and target
  - Process using Spark within Glue or pushdown processing to Snowflake
- ETL – Multiple sources
  - Read data from Snowflake and join with Data from S3 (and/or other sources)
  - Write back to Snowflake (using pushdown)
- Understand data assets – Data Catalog
“Serverless computation is going to fundamentally not only change the economics of what is back-end computing, but it’s going to be the core of the future of distributed computing.” —

2:47 PM - Dec 23, 2017

66 people love this
41 people are talking about this
THANK YOU

Going Serverless - an Introduction to AWS Glue
Reach out for more info!

- Email: michael.rainey@snowflake.com
- Twitter: @mRainey

AWS Glue resources

- Docs:

- Forums:

- Presentations:
  - [Serverless ETL with AWS Glue](http://aws.amazon.com/documentation/glue)
  - [How to Build a Data Lake with AWS Glue Data Catalog](http://aws.amazon.com/documentation/glue)

- Blogs:
  - [How To Use AWS Glue With Snowflake](http://aws.amazon.com/documentation/glue)