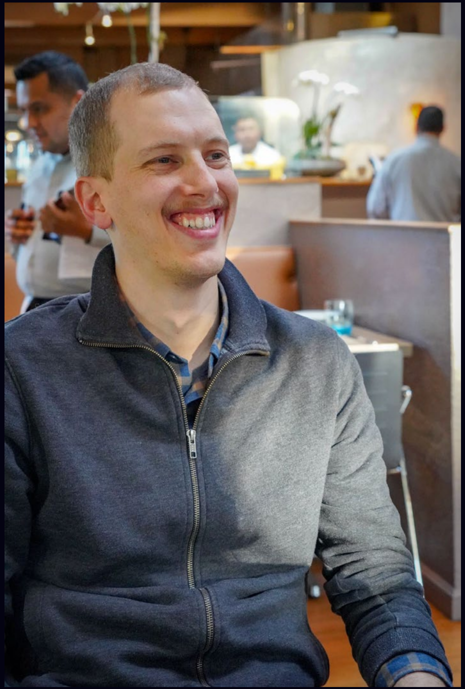
Fast, Cheap, and Easy Data Ingestion
with AWS Lambda and Delta Lake

About me

Howdy! My name is **R. Tyler Croy**

- I helped create the **delta-rs** project.
- I write lots of Rust.
- I authored a chapter in **Delta Lake: The Definitive Guide**.
- I help organizations build cloud-native data platforms.
- I can help you lower the cost of your Databricks *and* AWS bills!

artist's rendering



Let's define our terms

Delta Lake

- Data storage format which is basically:
    - JSON transaction **log files**
    - Apache Parquet **data files**
- In AWS we store Delta tables in **S3**

```
s3://bucket/delta-table

 ├── ds=2024-04-01
│    ├── part-00000-d361a60627e3.c000.snappy.parquet
│   └── part-00001-5d1872324d6f.c000.snappy.parquet
 ├── ds=2024-04-02
│    ├── part-00000-de0b22b62bbd.c000.snappy.parquet
│   └── part-00001-25f7559cd150.c000.snappy.parquet
└── _delta_log
   └── 00000000000000000000.json
```

AWS Lambda

> AWS Lambda is an event-driven, serverless Function
> as a Service. It is designed to enable developers
> to run code without provisioning or managing
> servers. It executes code in response to events
> and automatically manages the computing resources
> required by that code

Lambda supports multiple run times but most important are its Python and rust support for our discussion

- `pip install cargo-lambda`

Time is money

Lambda charges based on:

- **Execution Time**: *faster is cheaper*
- **Memory used**: *smaller is better*
- **Storage**: *not really important*

▒ Serverless data processing

everything we'll discuss can be done in other serverless environments so long as they support:

- event notifications
- triggered execution
- object storage

Could easily be converted to run on Azure or Google Cloud functions

```
def lambda_handler(event, context):
        from deltalake import DeltaTable
   dt = DeltaTable(os.environ['TABLE_URL'])
   metadata = dt.metadata()
        return {
      'version' : dt.version(),
      'table' : url,
      'files' : dt.files(),
   'metadata' : {
         'name' : metadata.name,
         'created_time' : metadata.created_time,
         'id' : metadata.id,
         'description' : metadata.description,
         'partition_columns' : metadata.partition_columns,
         'configuration' : metadata.configuration,
      },
   }
```

Lambda

```
async fn function_handler(_event: Request) -> Result<Response<Body>, Error> {
     let table_url = std::env::var("TABLE_URL")
    .expect("must set TABLE_URL in env");

     let response = match deltalake_core::open_table(&table_url).await {
    Ok(table) => {
               let files: Vec<String> = table
        .get_files_iter()?
        .map(|p| p.as_ref().to_string())
        .collect();

               let message = json!({
          "table" : table_url,
          "version" : table.version(),
          "metadata" : table.metadata()?,
          "files" : files,
      });
      Response::builder()
        .status(200)
        .header("content-type", "application/json")
        .body(message.to_string().into())
        .map_err(Box::new)?
    }
    Err(e) => Response::builder()
      .status(500)
      .header("Content-Type", "text/plain")
      .body(format!("error: {e:?}").into())
      .map_err(Box::new)?,
  };

  Ok(response)
}
```

▨ S3 Event Notifications

S3 event notifications can be configured on a per event basis so that you only get objects created objects deleted or objects updated with new versions or any combination thereof

event notifications can be configured on a per prefix basis

but there can only be **one type of events notification** per prefix and suffix combo

Non-S3 event notifications

for a lot of the workloads that I manage I rely on S3 event notifications but when I need to ingest data from other sources I prefer to trigger Lambda from sqs

- eventbridge
- http payloads
- ???

SQS gives us dead letter queues and replay in case of failures which is important for a fault tolerant data ingestion process

oxbow

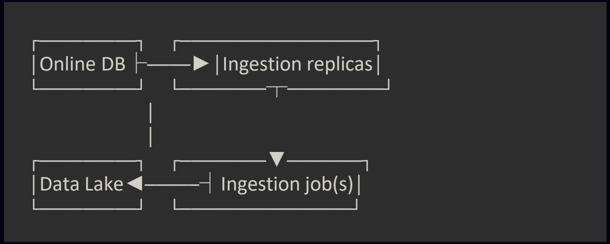
https://github.com/buoyant-data/oxbow

Oxbow started as a single Lambda that built off of S3 Event Notifications and has grown into a collection of tools and lambdas to manage Delta Lake

- oxbow
- group-events
- glue-create
- glue-sync
- webhook
- sqs-ingest

Case Study: Aurora export

Problem: a key component of most data processing systems is ingesting online data. this can entail running replicas for ingestion or expensive spark jobs to copy data out of live database systems.



Using Cloud native databases like AWS Aurora which now support direct export to AWS S3 in parquet format Delta tables can be reconstructed directly from the exported data without requiring any replicas or data processing

Case Study: Aurora export

```
┌──────────┐     ┌────────────┐
│Online DB ├────▶│ S3 bucket  │
└──────────┘     └─────┬──────┘
                       │
              ┌────────▼───┐
              │restructure │
              │lambda      │
              └───┬────────┘
                  │
              ┌───▼───┐
              │  SQS  │
              └───┬───┘
                  │
┌───────────┐ ┌───▼──────┐
│ Data Lake │◀┤  Oxbow   │
└───────────┘ └──────────┘
```

▒▒▒ Case Study: Aurora export

```
┌──────────┐      ┌────────────┐
│Online DB ├──────▶│ S3 bucket │
└──────────┘      └────────────┘
```

there are two ways we can get parquet out of an aurora database:

- direct export of parquet to S3
- through snapshots which export to parquet

Case Study: Aurora export

```
┌───────────┐
│ S3 bucket │
└─────┬─────┘
      │
┌─────▼─────┐
│restructure│
│lambda     │
└───────────┘
```

once the actual export has occurred the parquet files need to be restructured into the file layout that we expect in our Delta table this is a good time to do a `ds=` partition or lay the files out in any way that is desired

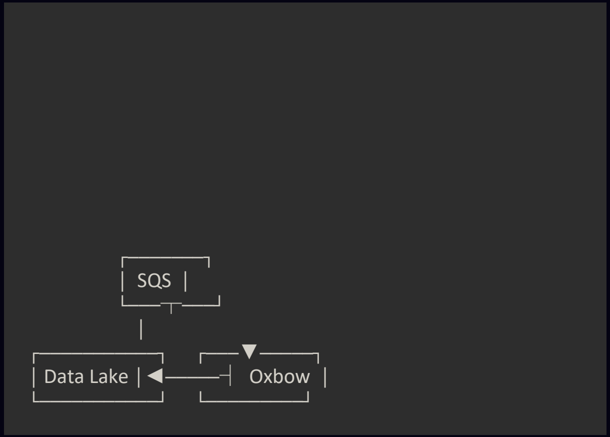
Case Study: Aurora export

```
┌────────────┐
│restructure │
│lambda      │
└───┬────────┘
    │
┌───▼───┐
│  SQS  │
└───────┘
```

restructuring the parquet files typically means copying and re-putting them to the same bucket or a different bucket which can then trigger additional S3 event notifications

these event notifications contain the file size and the key inside of the bucket

Case Study: Aurora export



one of the nice things about the deltalake crate and how Oxbow uses it is that it never has to read or understand the data files when the event notification is received from sqs it is translated into effectively in append on the Delta Lake transaction log

Caveats: Aurora export

```
┌──────────┐    ┌────────────┐
│Online DB ├───▶│ S3 bucket  │
└──────────┘    └─────┬──────┘
                      │
          ┌───────────▼───────┐
          │restructure│
          │lambda     │
          └─────┬─────────────┘
                │
          ┌─────▼───┐
          │  SQS    │
          └─────┬───┘
                │
┌──────────┐   ┌──▼──────────┐
│ Data Lake│◀──┤  Oxbow      │
└──────────┘   └─────────────┘
```

there are some caveats to this approach however:

- schema evolution is not yet supported because Oxbow doesn't know anything about the actual data file
- additionally there can be opportunities for lock contention
- some orchestration may be required if Downstream pipelines need to know exactly when a table has fully completed its export

Benefits: Aurora export

- $$$
- generally speaking this approach is cheaper and simpler than its predecessor using spark and replicas for Ingestion
- the Oxbow Lambda is about 5 MB and it's run time for most cases is so infrequent and quick that it fits into the AWS Lambda free tier which effectively means the cost of ingestion is the cost of storage in S3 and the snapshot operations from Aurora

Oxbow is an example written in Rust which is lots of fun but not for everybody

let's talk about **python**

▨ Python Lambdas

Distributing a Lambda with python typically means uploading a zip file or creating a layer or creating a Docker image

- .zip a few MBs
- layer: 250MBs total
- image: 10GB

▨ Python + deltalake Lambda

a rust example using the Lambda runtime and the Delta Lake crate along with data Fusion typically will produce a binary less than 10 MB in size

with python we need to pull in the Delta Lake package and typically will want the AWS SDK for pandas layer as well

in almost all cases that I have tried that pushes you over the limit of what a layer can contain

save yourself the headache and start with a Docker image

Pandas and Polars

loading a Delta table and then turning that into a panda's data frame is very straightforward but because of the memory constraint of a Lambda you have to provide additional filters otherwise you will blow up memory

```
dt = DeltaTable(table_url)
df = dt.to_pandas(filters=[('ds', '=', '2024-06-01')])
```

▨ failure modes

- lambdas can run out of memory easily
- can also time out if the runtime is longer than expected
- concurrency can be a challenge when operating on the same table

▨ Q&A

thanks!

buoyantdata.com