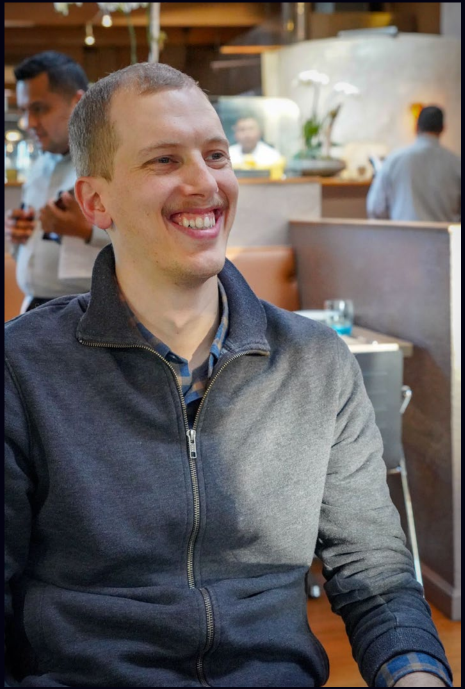
▓▓▓ Data Engineering with
▓▓▓ Rust and Delta Lake

🟦 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
```

Delta Lake

```
cat deltatbl-partitioned/_delta_log/00000000000000000000.json
```

Rust

> Rust is a multi-paradigm, general-purpose programming language that emphasizes performance, type safety, and concurrency. It enforces memory safety—meaning that all references point to valid memory—without a garbage collector

there are a lot of different ways to use rest for data engineering and processing but the big reason we want it is because it allows us to correctly Implement high performance programs with **less work**

our tools

- arrow
- deltalake
- datafusion
- *and more*

our tools: arrow

arrow is the foundation for almost all consequential data processing in Rust.

the big things that the arrow-rs project gives us are the in-memory columnar data representation of RecordBatch and a parquet reader/writer library

```
let arrow_array: Vec<Arc<dyn Array>> = vec![
        Arc::new(TimestampMicrosecondArray::from(ts)),
        Arc::new(Int32Array::from(temp)),
        Arc::new(Float64Array::from(lat)),
        Arc::new(Float64Array::from(long)),
];

RecordBatch::try_new(arrow_schema_ref, arrow_array)
   .expect("Failed to create RecordBatch")
```

our tools: arrow

working with arrow directly is typically a little more difficult than most people want so serde_arrow library helps and a couple other ways to generate RecordBatch structs

https://github.com/chmp/serde_arrow

▒ our tools: deltalake

`cargo add --features datafusion deltalake`

- metapackage contains:
  - deltalake-aws
  - deltalake-azure
  - deltalake-gcp

```
#[tokio::main]
async fn main() {
  deltalake::aws::register_handlers(None);
      let dt = deltalake::open_table("s3://bucket/table")
        .await
        .expect("Failed to open");
      // ....
}
```

our tools: datafusion

> DataFusion is a very fast, extensible query
> engine for building high-quality data-centric
> systems in Rust, using the Apache Arrow
> in-memory format.

just about every rust data project uses datafusion in some form or another

- `datafusion::DataFrame`
- Datafusion SQL

async/await

```
async fn main() -> Result<(), deltalake::errors::DeltaTableError> {
        let table_path = "../test/tests/data/delta-0.8.0";
        let table = deltalake::open_table(table_path).await?;
   println!("{table}");
   Ok(())
}
```

▒▒▒ references!

Rust is very strict about references:

- `&foo` cannot be sent between threads safely
- `Arc<Foo>` can be read safely between threads
- `Arc<Mutex<Foo>>` can be read and modified between threads

▒ **let's engineer some data**

building..

```
cargo new --bin uniproc
cd uniproc
ls
```

```
cd uniproc
cargo run
```

▨ building..

we'll need deltalake with its rich integration with DataFusion

```
cd uniproc
cargo add --features macros tokio
cargo add --features datafusion deltalake
cat Cargo.toml
```

▒ building..

```
        let ctx = SessionContext::new();
        let table = deltalake::open_table("../deltatbl-partitioned")
                .await?;
ctx.register_table("demo", Arc::new(table))?;

        let batches = ctx
                .sql("SELECT * FROM demo LIMIT 3").await?
                .collect()
                .await?;
        print_batches(&batches).expect("Failed to print batches");
```

```
cp sql-main.rs uniproc/src/main.rs
cd uniproc && cargo run
```

🟩 appending!

DataFrame is very powerful.

With the deltalake APIs if you can get a RecordBatch you can do almost anything.

```
let df = ctx.read_csv("../example.csv",
  CsvReadOptions::new()).await?;
let table = DeltaOps::from(table)
  .write(df.collect().await?)
  .await?;

ctx.register_table("demo", Arc::new(table))?;
let batches = ctx
  .sql("SELECT * FROM demo LIMIT 3").await?
  .collect()
  .await?;
```

appending!

```
cat example.csv
```

appending!

```
tree deltatbl-partitioned
```

appending!

```
rm -rf deltatbl-partitioned-write
cp -R deltatbl-partitioned deltatbl-partitioned-write
cp write-main.rs uniproc/src/main.rs
cd uniproc && cargo run
```

appending!

```
tree deltatbl-partitioned-write
```

other operations

```rust
use deltalake::DeltaOps;
```

- Merge
- Update
- Optimize
- ZOrder
- Vacuum

kafka-delta-ingest

- ingests avro and json
- utilizes txn action for state tracking
- should be deployed 1 per topic:partition

```
{"commitInfo":{   }}
{"add":{"path":"f3",   }}
{"add":{"path":"f4",   }}
{"txn":{"appId":"3ae45b72","version":4389}}
```

arroyo

```
CREATE TABLE delta_sink (
    id INTEGER,
    name STRING,
    age INTEGER
) WITH (
    'connector' = 'delta',
    'path' = 's3://my_bucket/my_table',
    'format' = 'parquet',
    'filename.strategy' = 'uuid'
);
INSERT INTO delta_sink SELECT id, name, age FROM my_source;
```

see also

- roapi
- ParadeDB
- Apache Comet

░ thanks!

Q&A

buoyantdata.com