DATA+AI SUMMIT BY databricks

HOW TO USE DELTA SHARING FOR STREAMING DATA

Matt Slack - Senior Specialist Solution Architect
Surya Sai Turaga - Senior Solution Architect
13th June 2024

Product safe harbor statement

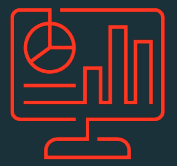
This information is provided to outline Databricks' general product direction and is for **informational purposes only**. Customers who purchase Databricks services should make their purchase decisions relying solely upon services, features, and functions that are currently available. Unreleased features or functionality described in forward-looking statements are subject to change at Databricks discretion and may not be delivered as planned or at all

Customer Story

- **Company** - data provider for financial market data (similar to Bloomberg/State Street)
- **Use Case -** create a marketplace for financial data products, e.g.:
  - pricing and market data
  - company data
  - risk intelligence
  - economic data
  - **news**
  - commodities data
- **Success Criteria**
  - consistent sub 10 second data delivery to customers
  - integrates into customers existing systems
  - requires minimal additional infrastructure

When is streaming for Delta Sharing a good fit?

Make your data available anywhere *with low latency*



Supports medium latency

- Achievable latency currently around 10s
- Supports high volumes of data
- Enables many other use cases

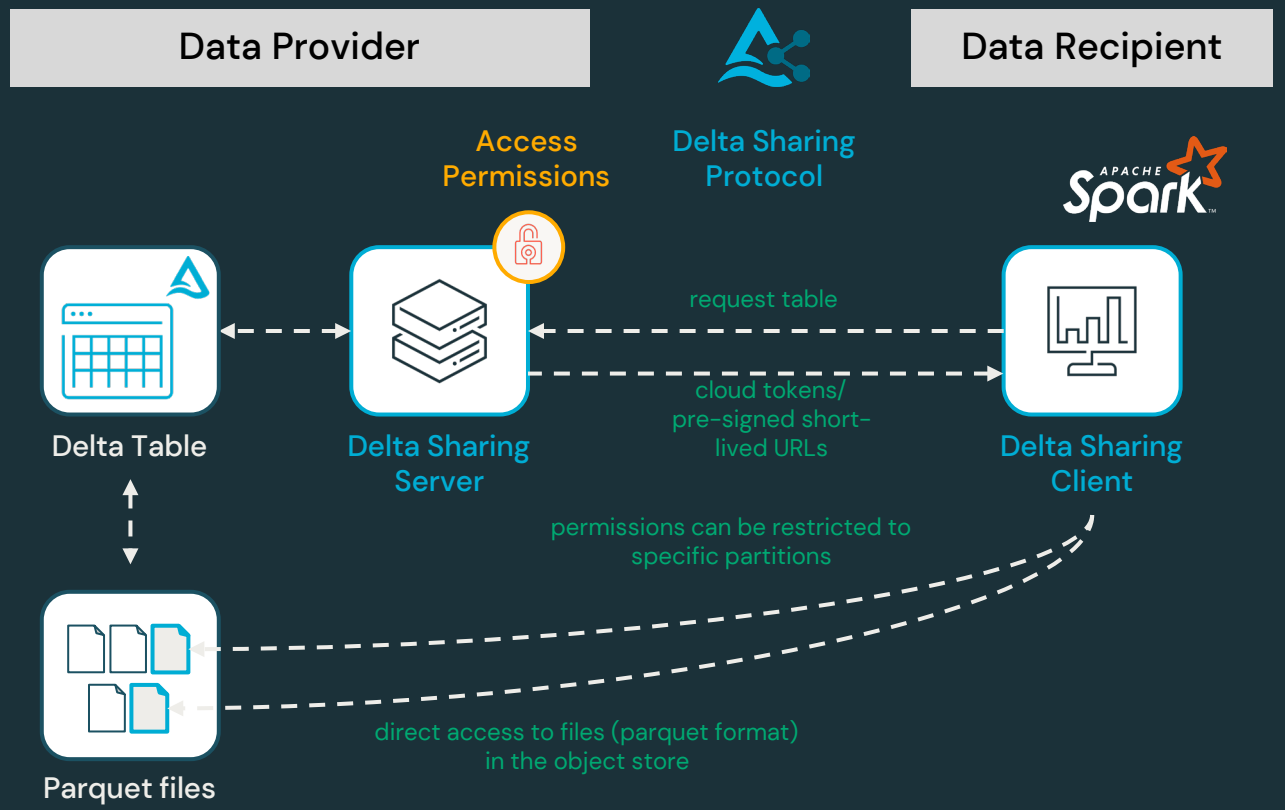
Reduces complexity

- No additional infrastructure needed (Kafka, Eventhubs, Kinesis etc.)
- No (de)serialization to AVRO/Protobuf/JSON...
- Schema management without a schema registry

Sharing externally and cross-cloud

- Common semantics for accessing tables with batch/stream
- Secure for cross cloud data sharing

Recap Delta Sharing architecture




```
spark.read.format("deltaSharing")
```

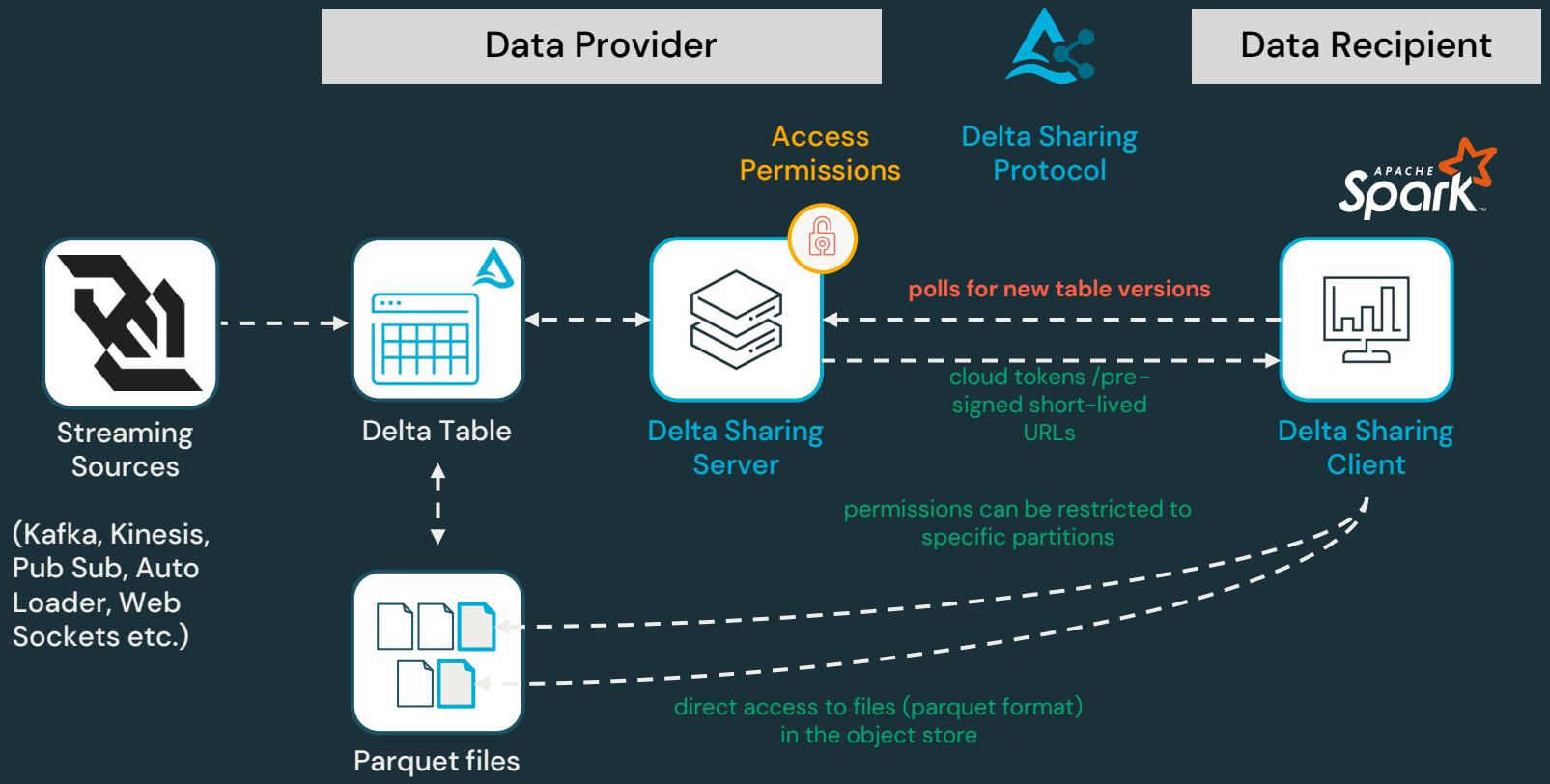
Delta Sharing Protocol:

- Client authenticates to Sharing Server
- Client requests a table (including filters)
- Server checks access permissions
- Server generates and returns cloud tokens/pre-signed short-lived URLs
- Client uses cloud tokens/URLs to directly read files from object storage

Notes:

- Sharing happens on Delta part files, supporting full tables, partitions, delta versions, ...
- Client is system independent, just needs to be able to read parquet files
- In Databricks Sharing Server and ACL checks are integrated with Unity Catalog

Streaming with Delta Sharing




```
spark.readStream.format("deltaSharing")
```

Delta Sharing Protocol:

- Client authenticates to Sharing Server
- **Client polls for new table versions**
- Client requests a **table version** (including filters)
- Server checks access permissions
- Server generates and returns cloud token/pre-signed short-lived URLs
- Client uses cloud token/URLs to directly read files from object storage **that correspond to that Delta table version**

Notes:

- **Client maintains current table version in the checkpoint directory (same as when streaming from a Delta table)**

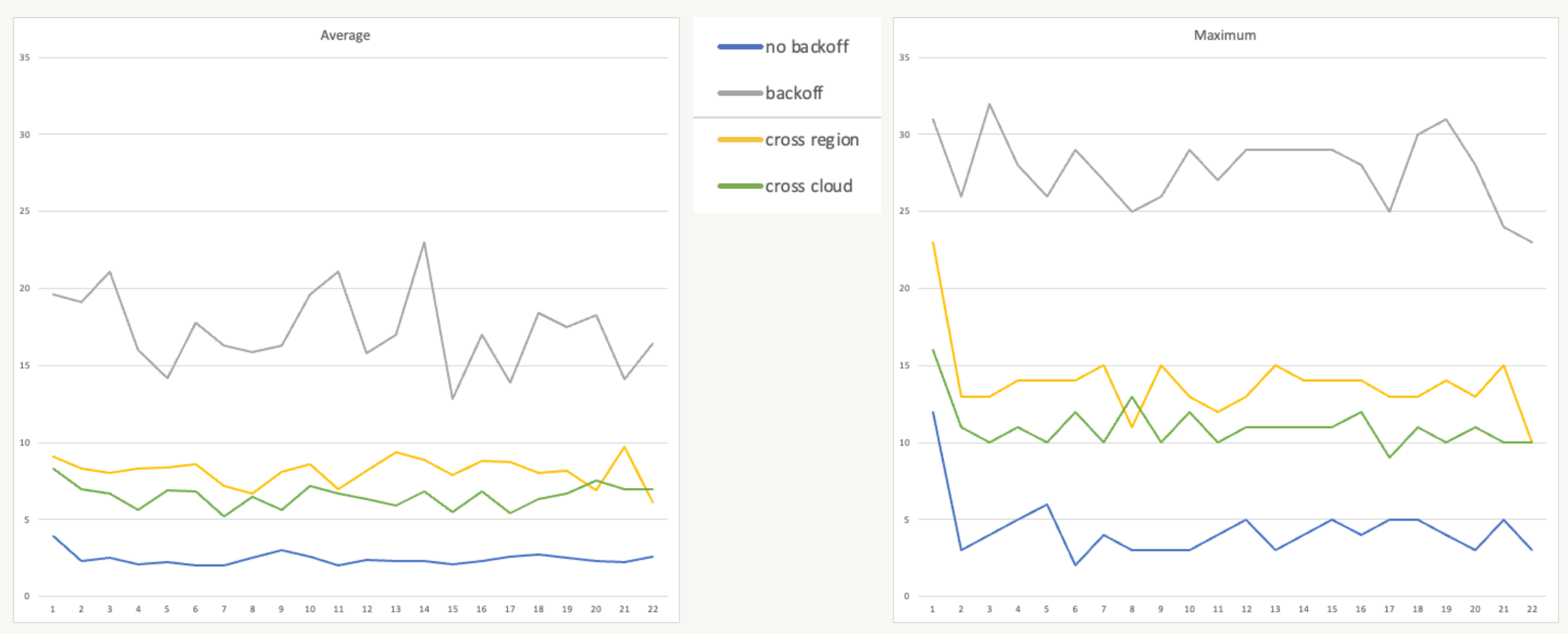
Optimising streaming read latency

Default client behavior is to throttle to prevent overloading the server

- Delta Sharing server is a shared resource, so streaming client has built in throttling – can be reduced for low latency use cases `spark.delta.sharing.streaming.queryTableVersionIntervalSeconds`
- Calls to the Delta Sharing server can require unpacking the delta log which may take a few seconds, depending on cloud provider access times
- Partitioned tables will increase the number of files for each Delta table version, so more overhead for the Delta Sharing server

What latency is achievable?

Looking at different settings to improve throughput/latency



Monitoring streaming client progress

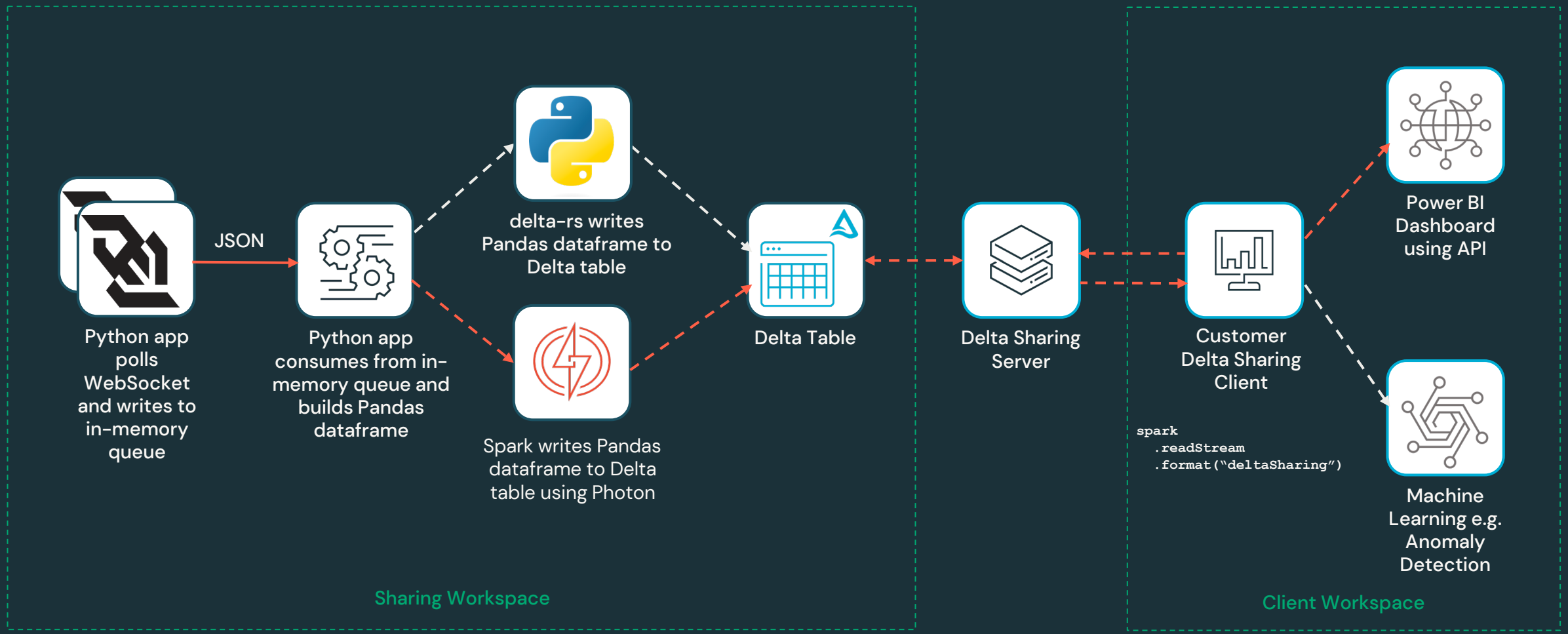
How can we ensure all clients have read a given table version

```
SELECT
  event_time,
  request_params.recipient_name recipient_name,
  CAST(GET_JSON_OBJECT(response.result, "$.scannedAddFileSize") AS INT) file_size,
  GET_JSON_OBJECT(response.result, "$.tableFullName") table_full_name,
  CAST(GET_JSON_OBJECT(response.result, "$.numRecords") AS INT) num_records,
  GET_JSON_OBJECT(response.result, "$.deltaSharingShareName") share_name,
  GET_JSON_OBJECT(response.result, "$.tableVersion") table_version
FROM
  system.access.audit
WHERE
  service_name = 'unityCatalog'
  AND action_name = 'deltaSharingQueriedTable'
  AND event_time > current_date()
  AND GET_JSON_OBJECT(response.result, "$.tableFullName") RLIKE
'.*dais_streaming_demo.*'
ORDER BY
  event_time DESC
```

- One row per client streaming query
- Shows all versions consumed
- Allows identification of which versions can be VACUUMed

Streaming Demo

Demo Architecture



Maintaining consistent write SLAs

delta-rs allows low-level tuning of Delta table writes

- Spark Delta writer can cause ~10s spikes in write times - exceeding customer SLAs
- Switching to delta-rs provides more control over the delta log
- Some caveats
  - delta-rs does **not** respect `delta.checkpointInterval`
  - delta-rs does **not** implement auto compaction
- Each append adds a new entry in `_delta_log` - ADLS file listing API slows down as the number of entries in `_delta_log` increases
- Write time increases by ~3s for every hour
- Manually checkpoint, OPTIMIZE and VACUUM the table regularly to speed-up directory listing
- PR to add `cleanup_metadata` to the Python API from the Rust API - allows clean-up of files in `_delta_log`

```
import deltalake as dl
dt = dl.DeltaTable(path, storage_options=storage_cfg)

configuration={
  "delta.logRetentionDuration": "interval 5 minute",
  "delta.enableExpiredLogCleanup": "true"}

dl.write_deltalake(dt, data=message_pd, mode="append",
                   storage_options=storage_cfg,
                   partition_by=partition_cfg,
                   configuration=cfg)

# run this every 25 batches
dt.create_checkpoint()
dt.cleanup_metadata() # new method added to Python API

# run this every 100 batches
dt.optimize.compact()
```


Questions