Massive Data Processing in Adobe using Delta Lake

Yeshwanth Vijayakumar
Sr. Engineering Manager/Architect @ Adobe
Agenda

- Introduction
- What are we storing?
- Data Representation and Nested Schema Evolution
- Writer Worries and How to Wipe them Away
- Staging Tables FTW
- Datalake Replication Lag Tracking
- Performance Time!
Unified Profile Data Ingestion

Adobe Campaign
AEM
Adobe Analytics
Adobe AdCloud

Experience Data Model

Unified Profile
Single Tenant
Change Feed
Multi Tenant
Streaming Stats Generation
Linking Identities

### Anonymous Identities

<table>
<thead>
<tr>
<th>IP Address</th>
<th>ECID</th>
<th>AMO ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>192.18.86.50</td>
<td>GT8ERW8tOK70g</td>
<td>kPL1</td>
</tr>
<tr>
<td>192.18.86.50</td>
<td>ZXBEUW8tLP7Qn</td>
<td>POKN</td>
</tr>
</tbody>
</table>

### Known Identities

#### Online Data

- **Login ID**: arun@domain.com
  - ECID: ZXBEUW8tLP7Qn
  - Loyalty ID: 3690098
- **Email ID**: michelle@domain.com
  - ECID: GZYIB-h_hACHtIR
  - Loyalty ID: 5846890

#### Offline Data

- **Email ID**: arun@domain.com
  - Loyalty ID: 82167672165
- **Email ID**: rahul@domain.com
  - Loyalty ID: 34657616546
  - CRM ID: 2209
## Data Layout At a Glance

An Idea about how the graph linkages are stored

<table>
<thead>
<tr>
<th>primaryId</th>
<th>relatedIds</th>
<th>field1</th>
<th>field2</th>
<th>field1000</th>
</tr>
</thead>
<tbody>
<tr>
<td>123</td>
<td>123</td>
<td>a</td>
<td>b</td>
<td>c</td>
</tr>
<tr>
<td>456</td>
<td>456</td>
<td>d</td>
<td>e</td>
<td>f</td>
</tr>
<tr>
<td>123</td>
<td>123</td>
<td>d</td>
<td>e</td>
<td>l</td>
</tr>
<tr>
<td>789</td>
<td>789,101</td>
<td>x</td>
<td>y</td>
<td>z</td>
</tr>
<tr>
<td>101</td>
<td>789,101</td>
<td>x</td>
<td>u</td>
<td>p</td>
</tr>
</tbody>
</table>

**Conditions**
- primaryId does not change
- relatedIds can change
New Record comes in linking 103 with 789 and 101

<table>
<thead>
<tr>
<th>primaryId</th>
<th>relatedId</th>
<th>field1</th>
<th>field2</th>
<th>field1000</th>
</tr>
</thead>
<tbody>
<tr>
<td>103</td>
<td>103,789,101</td>
<td>q</td>
<td>w</td>
<td>r</td>
</tr>
</tbody>
</table>

Causes a cascading change in rows of 789 and 101

<table>
<thead>
<tr>
<th>primaryId</th>
<th>relatedId</th>
<th>field1</th>
<th>field2</th>
<th>field1000</th>
</tr>
</thead>
<tbody>
<tr>
<td>103</td>
<td>103,789,101</td>
<td>q</td>
<td>w</td>
<td>r</td>
</tr>
<tr>
<td>789</td>
<td>103,789,101</td>
<td>x</td>
<td>y</td>
<td>z</td>
</tr>
<tr>
<td>101</td>
<td>103,789,101</td>
<td>x</td>
<td>y</td>
<td>z</td>
</tr>
</tbody>
</table>
Main Access Pattern
Multiple Queries over 1 consolidated row

```javascript
rawRecords
  .groupBy("relatedIds")
  .mapPartitons{
    (relatedIds, records) => {
      results = executeQueries(records)
      saveResultsToSink(results, relatedIds)
    }
  }
```
Complexities?

- **Nested Fields**
  - a.b.c.d[*].e nested hairiness!
  - Arrays!
  - MapType

- **Every Tenant has a different Schema!**

- **Schema evolves constantly**
  - Fields can get deleted, updated.

- **Multiple Sources**
  - Streaming
  - Batch
Scale?

- Tenants have 10+ Billions of rows
- PBs of data
- Million RPS peak across the system
- Triggers multiple downstream applications
  - Segmentation
  - Activation
What is DeltaLake?

From delta.io: Delta Lake is an open-source project that enables building a Lakehouse architecture on top of existing storage systems such as S3, ADLS, GCS, and HDFS.

Key Features

- ACID Transactions
- Time Travel (data versioning)
- Uses Parquet Underneath
- Schema Enforcement and Schema Evolution
- Audit History
- Updates and Deletes Support
Delta lake in Practice

**UPsert**

dataframe
 .write
 .format("parquet")
 .save("/data")

dataframe
 .write
 .format("delta")
 .save("/data")

```
deltaTable.as("oldData")
 .merge(
   newData.as("newData"),
   "oldData.id = newData.id")
 .whenMatched
 .update(Map("id" -> col("newData.id")))
 .whenNotMatched
 .insert(Map("id" -> col("newData.id")))
 .execute()
```

**SQL Compatible**

```
UPDATE events SET eventType = 'click' WHERE eventType = 'clk'

UPDATE delta."/data/events/" SET eventType = 'click' WHERE eventType = 'clk'
```
Writer Worries and How to Wipe them Away

- **Concurrency Conflicts**

<table>
<thead>
<tr>
<th></th>
<th>INSERT</th>
<th>UPDATE, DELETE, MERGE INTO</th>
<th>COMPACTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>INSERT</td>
<td>Cannot conflict</td>
<td></td>
<td></td>
</tr>
<tr>
<td>UPDATE, DELETE, MERGE INTO</td>
<td>Can conflict</td>
<td>Can conflict</td>
<td></td>
</tr>
<tr>
<td>COMPACTION</td>
<td>Cannot conflict</td>
<td></td>
<td>Can conflict</td>
</tr>
</tbody>
</table>

- **Column size**
  - When individual column data exceeds 2GB, we see degradation in writes or OOM

- **Update frequency**
  - Too frequent updates cause underlying filestore metadata issues.
  - This is because every transaction on an individual parquet causes CoW,
    - More updates => more rewrites on HDFS

- **Too Many small files !!!**
CDC (existing)

Mutation Apps

1. Send Request to Cosmos

2. Ack

3. Emit CDC

CosmosDB

Consumed by
• Stats
• Edge
• etc

Batch Ingestion / Streaming Ingestion / API based Ingest
Dataflow with DeltaLake

<table>
<thead>
<tr>
<th>primaryId</th>
<th>relatedId</th>
<th>field1</th>
<th>field2</th>
<th>field1000</th>
</tr>
</thead>
<tbody>
<tr>
<td>103</td>
<td>103,789,101</td>
<td>q</td>
<td>w</td>
<td>r</td>
</tr>
<tr>
<td>789</td>
<td>103,789,101</td>
<td>x</td>
<td>y</td>
<td>z</td>
</tr>
<tr>
<td>101</td>
<td>103,789,101</td>
<td>x</td>
<td>y</td>
<td>z</td>
</tr>
</tbody>
</table>

Cosmos DB

Change Feed CDC

Long Running Streaming Application

Staging Table

APPEND only!
Partitioned by tenant and 15 min time intervals

Raw Table (per tenant)

<table>
<thead>
<tr>
<th>primaryId</th>
<th>relatedId</th>
<th>jsonString</th>
</tr>
</thead>
<tbody>
<tr>
<td>103</td>
<td>103,789,101</td>
<td>&lt;jsonStr&gt;</td>
</tr>
<tr>
<td>789</td>
<td>103,789,101</td>
<td>&lt;jsonStr&gt;</td>
</tr>
<tr>
<td>101</td>
<td>103,789,101</td>
<td>&lt;jsonStr&gt;</td>
</tr>
</tbody>
</table>

CDC Dumper

Backfill

TenantLock in Redis

Fetch Records to process

Processor

Check for Work every X minutes

UPSERT/DELETE into Raw Table
Staging Tables FTW

Fan-In pattern vs Fan-out

- **Multiple Source Writers Issue Solved**
  - By centralizing all reads from CDC, since ALL writes generate a CDC

- **Staging Table in APPEND ONLY mode**
  - No conflicts while writing to it

- **Filter out. Bad data > thresholds before making it to Raw Table**

- **Batch Writes by reading larger blocks of data from Staging Table**
  - Since it acts time aware message buffer
Staging Table Logical View

```xml
<TSKEY= 2021-01-01-09-15-Quarter=01 > -
  [ x1-cdcRecord, x2-cdcRecord, x3-cdcRecord, x5-cdcRecord ]

<TSKEY= 2021-01-01-09-15-Quarter=02 > -
  [ x2-cdcRecord, x7-cdcRecord ]

<TSKEY= 2021-01-01-09-15-Quarter=03 > -
  [ x6-cdcRecord, x9-cdcRecord ]
```

---

ProgressMap

<table>
<thead>
<tr>
<th>Org</th>
<th>Phase 1 LastSuccessfulTSKey</th>
</tr>
</thead>
<tbody>
<tr>
<td>tenant1</td>
<td>2021-01-01-09-15-Quarter=01</td>
</tr>
<tr>
<td>tenant2</td>
<td>2021-01-02-07-10-Quarter=04</td>
</tr>
<tr>
<td>tenant3</td>
<td>2021-01-01-11-19-Quarter=03</td>
</tr>
</tbody>
</table>
Why choose JSON String format?

- We are doing a lazy Schema on-read approach.
  - Yes. this is an anti-pattern.

- Nested Schema Evolution was not supported on update in delta in 2020
  - Supported with latest version

- We want to apply conflict resolution before upsert-ing
  - Eg. resolveAndMerge(newData, oldData)
  - UDF’s are strict on types, with the plethora of difference schemas , it is crazy to manage UDF per org in Multi tenant fashion
  - Now we just have simple JSON merge udfs
    - We use json-iter which is very efficient in loading partial bits of json and in manipulating them.

- Don’t you lose predicate pushdown?
  - We have pulled out all main push-down filters to individual columns
    - Eg. timestamp, recordType, id, etc.
  - Profile workloads are mainly scan based since we can run 1000’s of queries at a single time.
  - Reading the whole JSON string from datalake is much faster and cheaper than reading from Cosmos for 20% of all fields.
Schema On Read is more future safe approach for raw data

- Wrangling Spark Structs is not user friendly
- JSON schema is messy
  - Crazy nesting
  - Add maps to the equation, just the schema will be in MBs

- Schema on Read using Json-iter means we can read what we need on a row by row basis

- Materialized Views WILL have structs!
Partition Scheme of Raw records

- **RawRecords Delta Table**
  - recordType
  - dataSetId
  - **timestamp** (key-value records will use DEFAULT value)
  
  *z-order on primaryId*

*z-order - Colocate column information in the same set of files using locality-preserving space-filling curves*
1. `%%fs
2. `ls /tmp/test/4932497857C1DF40A49423C@AdobeOrg.raw.partitioned.delta`

<table>
<thead>
<tr>
<th>path</th>
<th>name</th>
<th>size</th>
</tr>
</thead>
<tbody>
<tr>
<td>/tmp/test/4932497857C1DF40A49423C@AdobeOrg.raw.partitioned.delta/</td>
<td>_delta_log/</td>
<td>0</td>
</tr>
<tr>
<td>/tmp/test/4932497857C1DF40A49423C@AdobeOrg.raw.partitioned.delta/</td>
<td>rt=<em>HIVE_DEFAULT_PARTITION</em></td>
<td>0</td>
</tr>
<tr>
<td>/tmp/test/4932497857C1DF40A49423C@AdobeOrg.raw.partitioned.delta/</td>
<td>rt=identity/</td>
<td>0</td>
</tr>
<tr>
<td>/tmp/test/4932497857C1DF40A49423C@AdobeOrg.raw.partitioned.delta/</td>
<td>rt=keyvalue/</td>
<td>0</td>
</tr>
<tr>
<td>/tmp/test/4932497857C1DF40A49423C@AdobeOrg.raw.partitioned.delta/</td>
<td>rt=timeseries/</td>
<td>0</td>
</tr>
</tbody>
</table>

Showing all 5 rows.

1. `%%fs
2. `ls adl://datalakeppdocjhd2.azuredatalakestore.net/core/profile/atlas/v1/4932497857C1DF40A49423C@AdobeOrg.raw.partitioned.delta/rt=timeseries/ek=5d64ec86b7469b16a8cf1295/`

<table>
<thead>
<tr>
<th>path</th>
<th>name</th>
<th>size</th>
</tr>
</thead>
<tbody>
<tr>
<td>adl://datalakeppdocjhd2.azuredatalakestore.net/core/profile/atlas/v1/4932497857C1DF40A49423C@AdobeOrg.raw.partitioned.delta/rt=timeseries/ek=5d64ec86b7469b16a8cf1295/</td>
<td>tsdate=50376-03-13/</td>
<td>0</td>
</tr>
</tbody>
</table>

Showing all 1 rows.
Replication Lag – 2 types

- **CDC Lag from Kafka**
  - Tells us how much more work we need to do to catch up to write to Staging Table

- **How we track Lag on a per tenant basis**
  - We track Max(TimeStamp) in CDC per org
  - We track Max(TSKEY) processed in Processor
  - Difference gives us rough lag of replication
Merge/UPSERT Performance

Live Traffic Use case: How long does it take X CDC messages to get upserted into Raw Table

<table>
<thead>
<tr>
<th>Action: UPSERT CDC stage into fragment</th>
<th>Time Taken</th>
</tr>
</thead>
<tbody>
<tr>
<td>170 K CDC Records – Maps to 100k Rows in Raw Table</td>
<td>15 seconds</td>
</tr>
<tr>
<td>1.7 Million CDC Records – Maps to 1 Million Rows in Raw Table</td>
<td>61 seconds</td>
</tr>
</tbody>
</table>

spark.sql("set spark.databricks.delta.autoCompact.enabled = true")
spark.sql("set spark.databricks.delta.optimizeWrite.enabled = true")
## Job Performance Time!

<table>
<thead>
<tr>
<th></th>
<th>Hot Store (NoSQL Store)</th>
<th>Delta Lake</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size of Data</td>
<td>1 TB</td>
<td>64 GB</td>
</tr>
<tr>
<td>Number of Partitions</td>
<td>80</td>
<td>189</td>
</tr>
<tr>
<td>Job Cores Used</td>
<td>112</td>
<td>112</td>
</tr>
<tr>
<td>Job Runtime</td>
<td>3 hours</td>
<td>25 mins</td>
</tr>
</tbody>
</table>
TakeAways

• Scan IO speed from datalake >>> Read from Hot Store
• Reasonably fast eventually consistent replication within minutes
• More partitions means better Spark executor core utilization
• Potential to aggressively TTL data in hot store
• More downstream materialization !!!
• Incremental Computation Framework thanks to Staging tables!