ONLINE , AROUND-THE-WORLD OPEN SOURCE DATABASE CONFERENCE

PERCONA

LIVEONLINE

MAY 12 - 13th

2021

Massive Data Processing in Adobe using Delta Lake

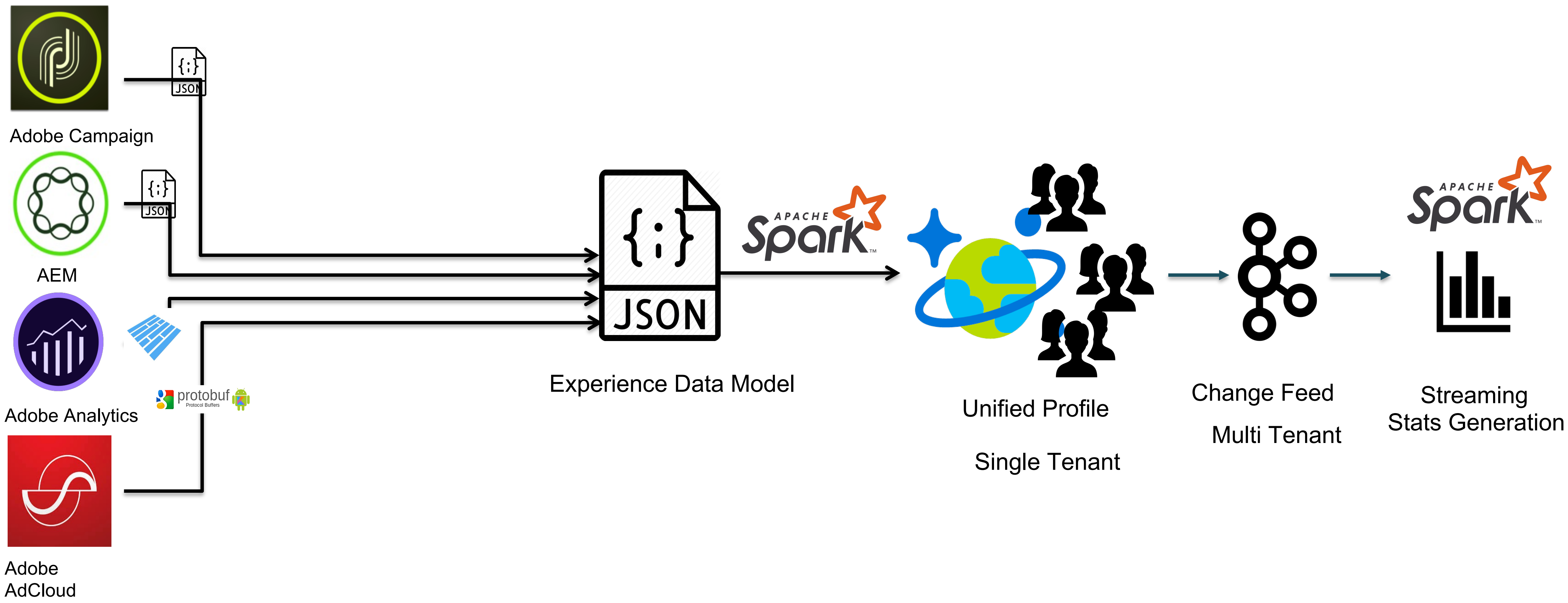
Yeshwanth Vijayakumar

Sr. Engineering Manager/Architect @ Adobe

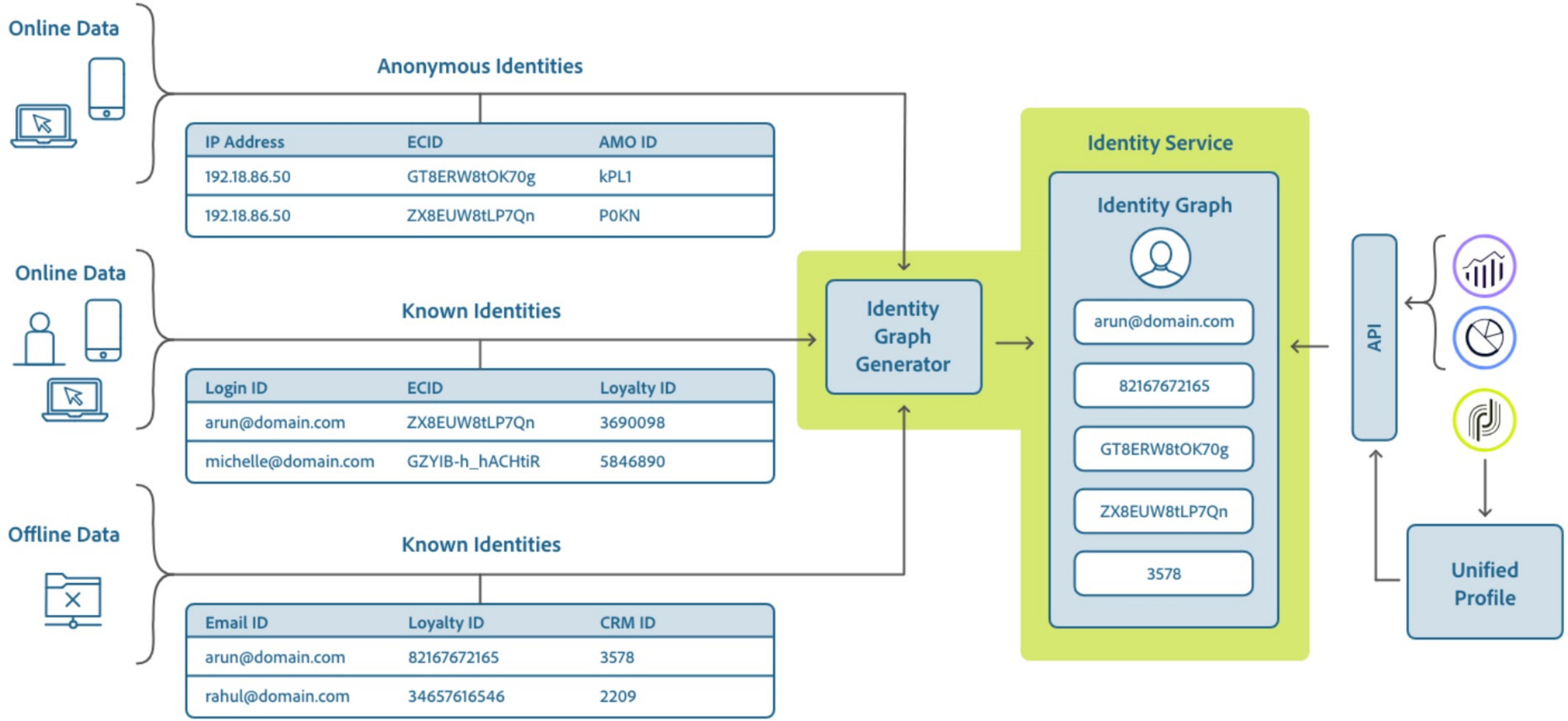
Agenda

- 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



Linking Identities



Data Layout At a Glance

An Idea about how the graph linkages are stored

Conditions
- primaryId does not change
- relatedIds can change

primaryId	relatedIds	field1	field2	field1000
123	123	a	b	c
456	456	d	e	f
123	123	d	e	l
789	789,101	x	y	z
101	789,101	x	u	p

New Record comes in

Indicating a new linkage, causing a change in graph membership

New Record comes in linking 103 with 789 and 101

primaryId	relatedId	field1	field2	field1000
103	103,789,101	q	w	r

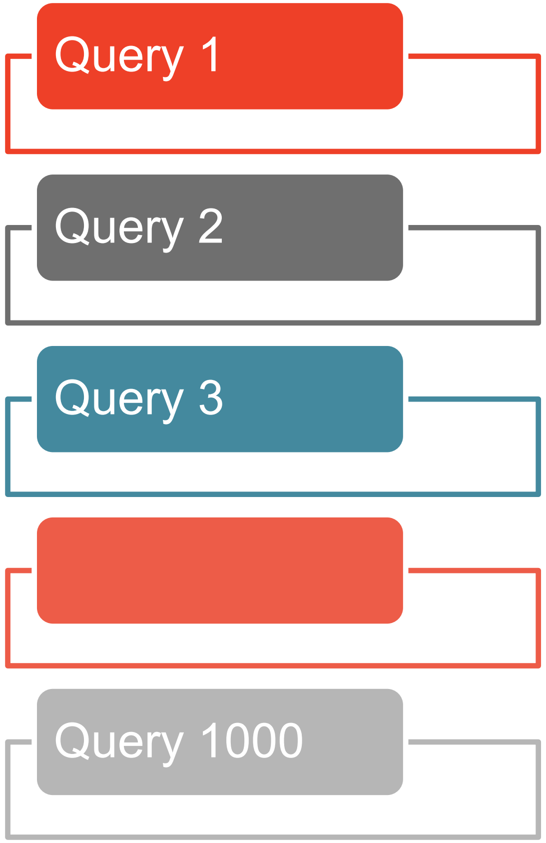
Causes a cascading change in rows of 789 and 101

primaryId	relatedId	field1	field2	field1000
103	103,789,101	q	w	r
789	103,789,101	x	y	z
101	103,789,101	x	y	z

Main Access Pattern

Multiple Queries over 1 consolidated row

```
rawRecords
  .groupBy($"relatedIds")
  .mapPartitons{
    (relatedIds, records) => {
      results = executeQueries(records)
      saveResultsToSink(results, relatedIds)

    }
  }
```



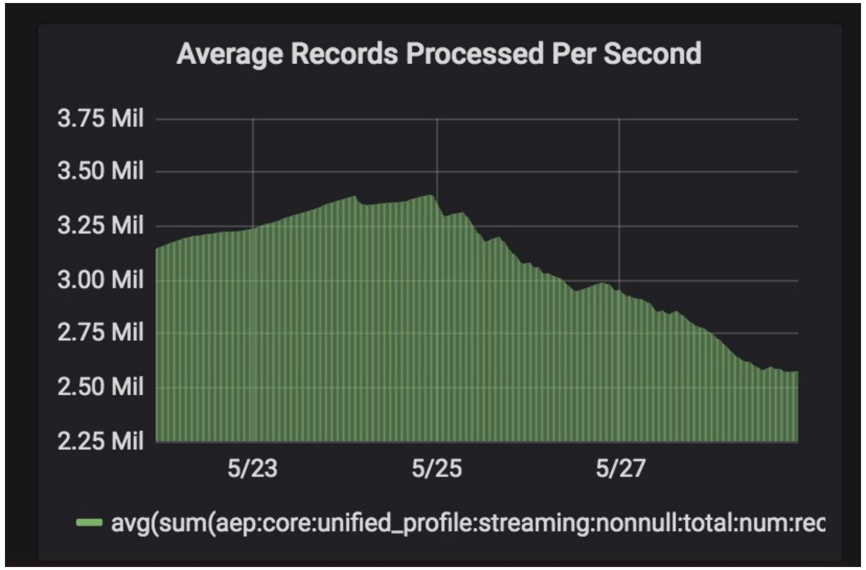
Complexities?

- 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


```
{
    "id":"xid1-source1",
    "ky":"GboEQWN4aWQx",
    "ek":"source1",
    "_ts":1539752290,
    "et":{
        "kv":{
            "identities":[
                {
                    "id":"luke@adobe.com",
                    "namespace":{
                        "code":"email"
                    }
                }
            ],
            "person":{
                "name":{
                    "firstName":"Luke",
                    "lastName":"Skywalker"
                }
            }
        }
    }
}
```

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?

DELTA LAKE

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

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

→

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

UPSERT

```
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 = 'clck'

UPDATE delta.`/data/events/` SET eventType = 'click' WHERE eventType = 'clck'
```

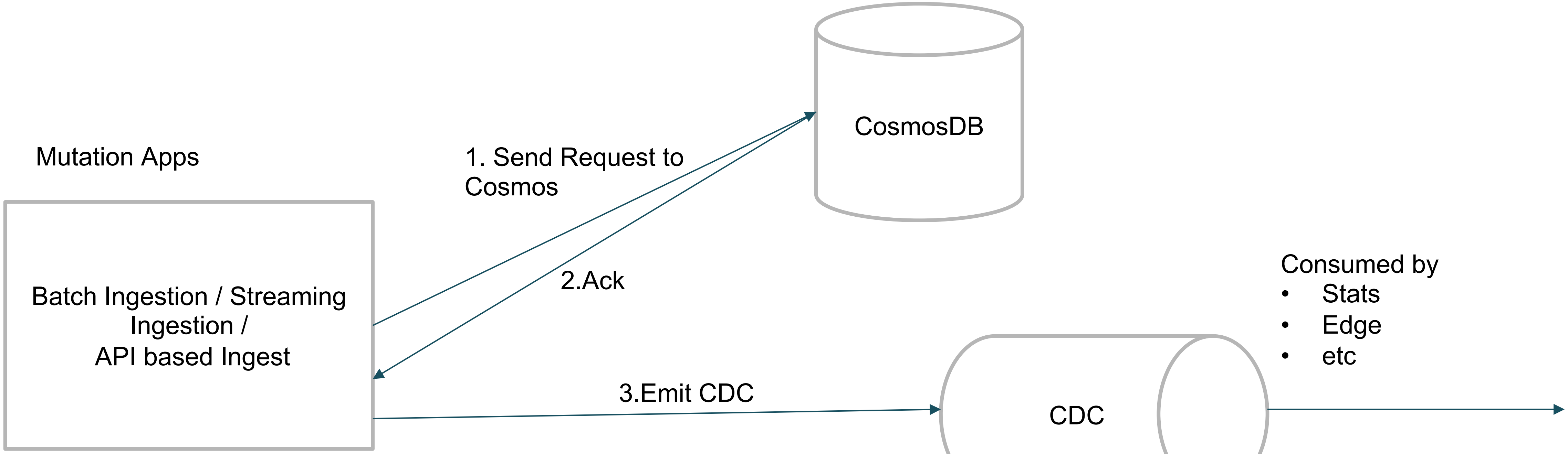
Writer Worries and How to Wipe them Away

- Concurrency Conflicts

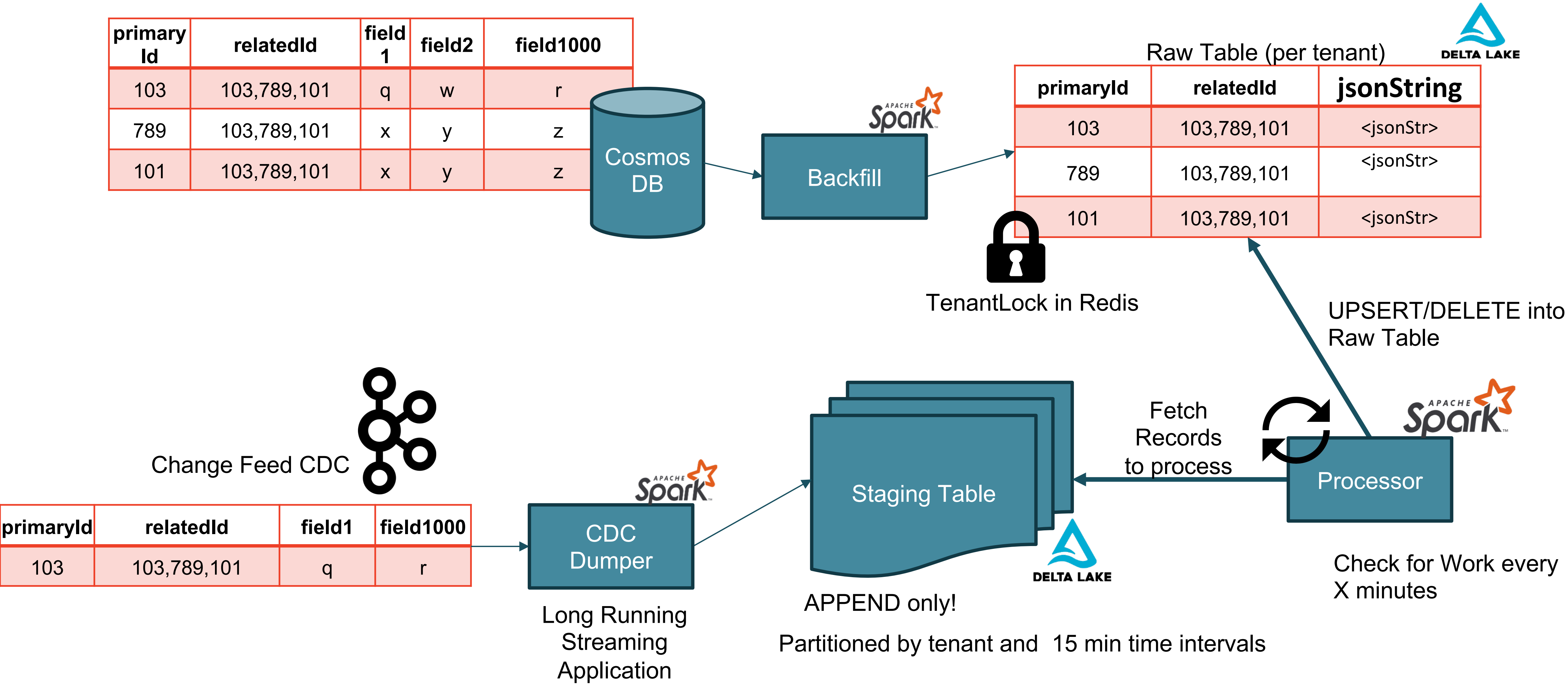
	INSERT	UPDATE, DELETE, MERGE INTO	COMPACTION
INSERT	Cannot conflict		
UPDATE, DELETE, MERGE INTO	Can conflict	Can conflict	
COMPACTION	Cannot conflict	Can conflict	Can conflict

- 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 transacation on an individual parquet causes CoW,
    - More updates => more rewrites on HDFS
  - **Too Many small files !!!**

CDC (existing)



Dataflow with DeltaLake



primary Id	relatedId	field 1	field2	field1000
103	103,789,101	q	w	r
789	103,789,101	x	y	z
101	103,789,101	x	y	z

Raw Table (per tenant)

primaryId	relatedId	jsonString
103	103,789,101	<jsonStr>
789	103,789,101	<jsonStr>
101	103,789,101	<jsonStr>

primaryId	relatedId	field1	field1000
103	103,789,101	q	r

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

```
<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

Org	Phase 1 LastSuccessfulTSKey
tenant1	2021-01-01-09-15-Quarter=01
tenant2	2021-01-02-07-10-Quarter=04
tenant3	2021-01-01-11-19-Quarter=03

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**

Cmd 42

```
%fs
ls /tmp/atlastest/4932D947587C1DF40A49423C@AdobeOrg.raw.partitioned.delta
```

Error runni

	path	name	size
1	dbfs:/tmp/atlastest/4932D947587C1DF40A49423C@AdobeOrg.raw.partitioned.delta/_delta_log/	_delta_log/	0
2	dbfs:/tmp/atlastest/4932D947587C1DF40A49423C@AdobeOrg.raw.partitioned.delta/rt=__HIVE_DEFAULT_PARTITION__/	rt=__HIVE_DEFAULT_PARTITION__/	0
3	dbfs:/tmp/atlastest/4932D947587C1DF40A49423C@AdobeOrg.raw.partitioned.delta/rt=identity/	rt=identity/	0
4	dbfs:/tmp/atlastest/4932D947587C1DF40A49423C@AdobeOrg.raw.partitioned.delta/rt=keyvalue/	rt=keyvalue/	0
5	dbfs:/tmp/atlastest/4932D947587C1DF40A49423C@AdobeOrg.raw.partitioned.delta/rt=timeseries/	rt=timeseries/	0

Showing all 5 rows.

Cmd 42

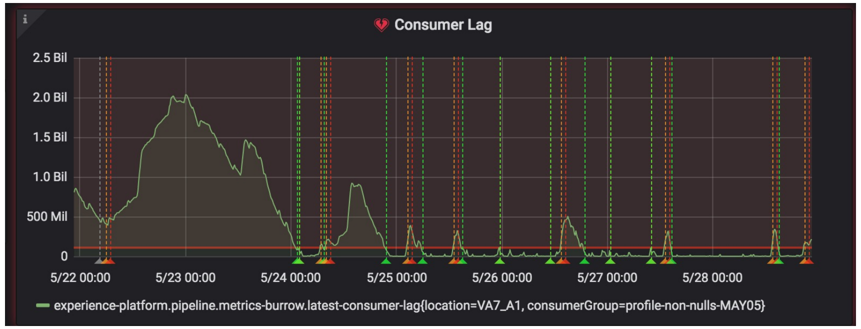
```
%fs
ls adl://datalakeeppvdoocjhd2.azuredatalakestore.net/core/profile/atlas/v1/4932D947587C1DF40A49423C@AdobeOrg.raw.partitioned.delta/rt=timeseries/ek=5d64ec86b7469b1648cf1295/

```

	path	name	size
1	adl://datalakeeppvdoocjhd2.azuredatalakestore.net/core/profile/atlas/v1/4932D947587C1DF40A49423C@AdobeOrg.raw.partitio	tsdate=50376-03-13/	0

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 Usecase: How long does it take X CDC messages to get upserted into Raw Table

Test cluster

Cluster
84G, 24Core, DBR 6.6, Standard_DS4_v2 (28.0 GB Memory, 8 Cores, 1.5 DBU), 1 driver, 2-8 workers

Action: UPSERT CDC stage into fragment	**Time Taken**
170 K CDC Records – Maps to 100k Rows in Raw Table	15 seconds
1.7 Million CDC Records – Maps to 1 Million Rows in Raw Table	61 seconds

```
spark.sql("set spark.databricks.delta.autoCompact.enabled = true")
spark.sql("set spark.databricks.delta.optimizeWrite.enabled = true")
```

Job Performance Time!

	Hot Store (NoSQL Store)	Delta Lake
Size of Data	1 TB	64 GB
Number of Partitions	80	189
Job Cores Used	112	112
Job Runtime	3 hours	**25 mins**

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!