

Online Meetup

Multi-Table Transactions with LakeFS and Delta Lake



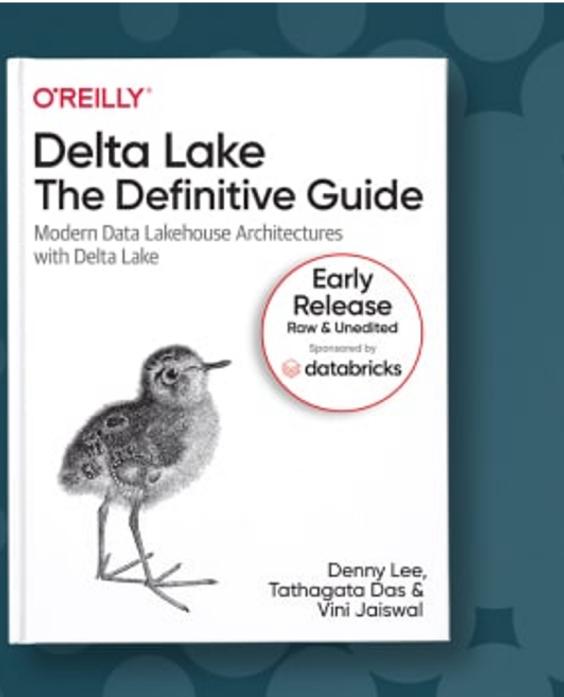
About the Speaker



Denny Lee is a Developer Advocate at Databricks. He is a hands-on distributed systems and data sciences engineer with extensive experience developing internet-scale infrastructure, data platforms, and predictive analytics systems for both on-premise and cloud environments.

O'Reilly Delta Lake: The Definitive Guide (*Early Release*)

Early Release
**O'Reilly
Definitive
Guide**



<https://dbricks.co/dldg>

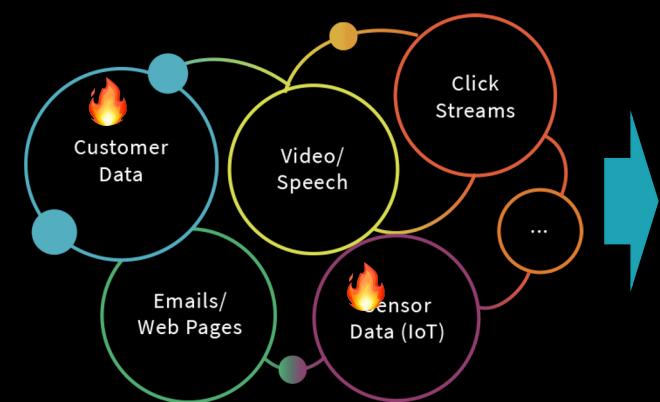
About the Speaker



Paul Singman is a Developer Advocate at lakeFS. He previously led a Platform Engineering team at Equinox Fitness, where he worked on operational and user-facing analytic services. When not staring at screens, he likes to run, golf, play ping-pong and backgammon.

The Promise of the Data Lake

1. Collect Everything



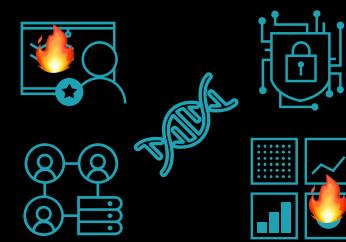
Garbage In

2. Store it all in the Data Lake



Garbage Stored

3. Data Science & Machine Learning



- Recommendation Engines
- Risk, Fraud Detection
- IoT & Predictive Maintenance
- Genomics & DNA Sequencing

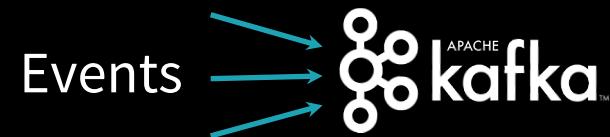
Garbage Out



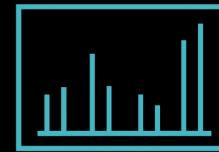
What does a typical
data lake project look like?



Evolution of a Cutting-Edge Data Lake



Data Lake



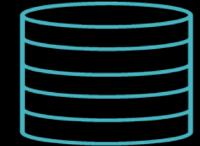
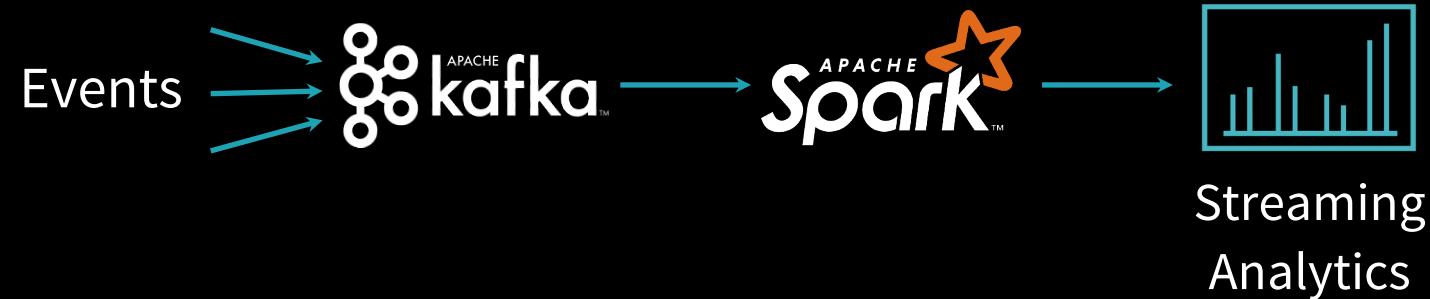
Streaming
Analytics



AI & Reporting



Evolution of a Cutting-Edge Data Lake



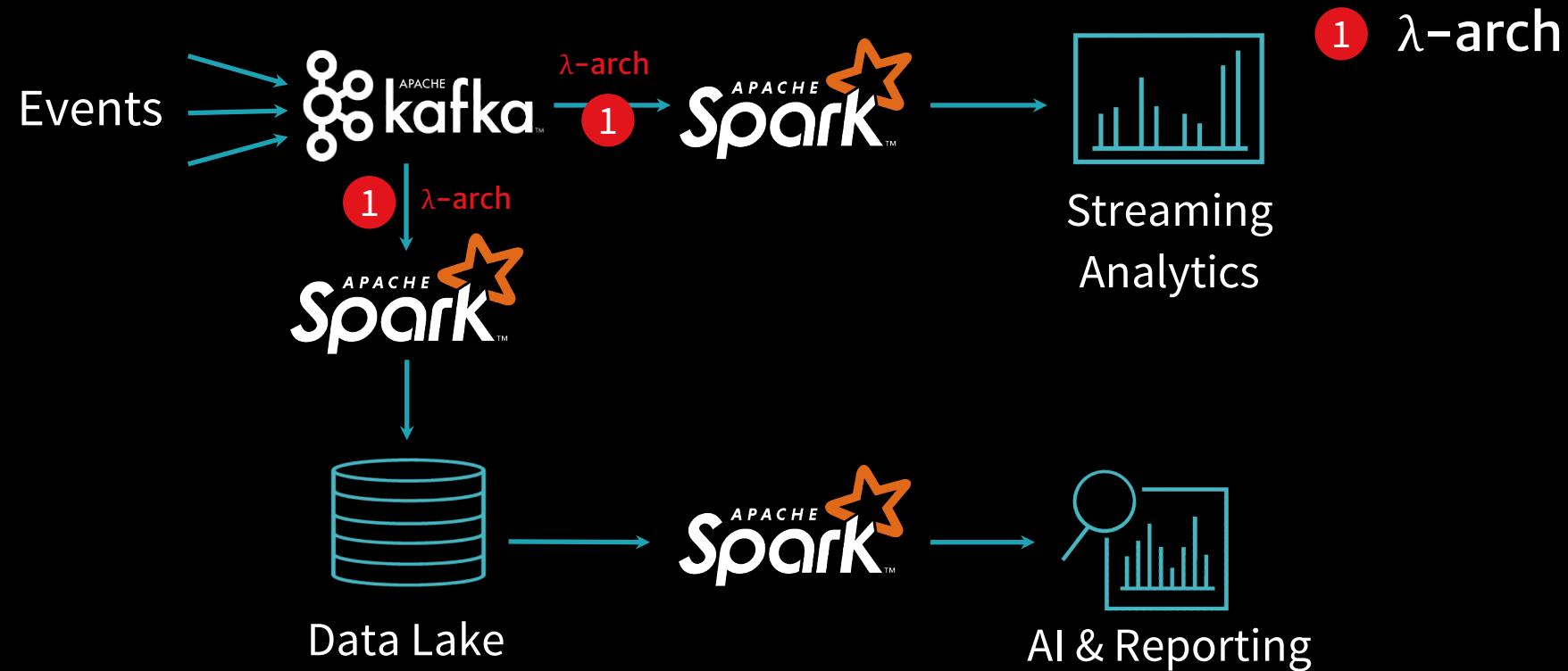
Data Lake



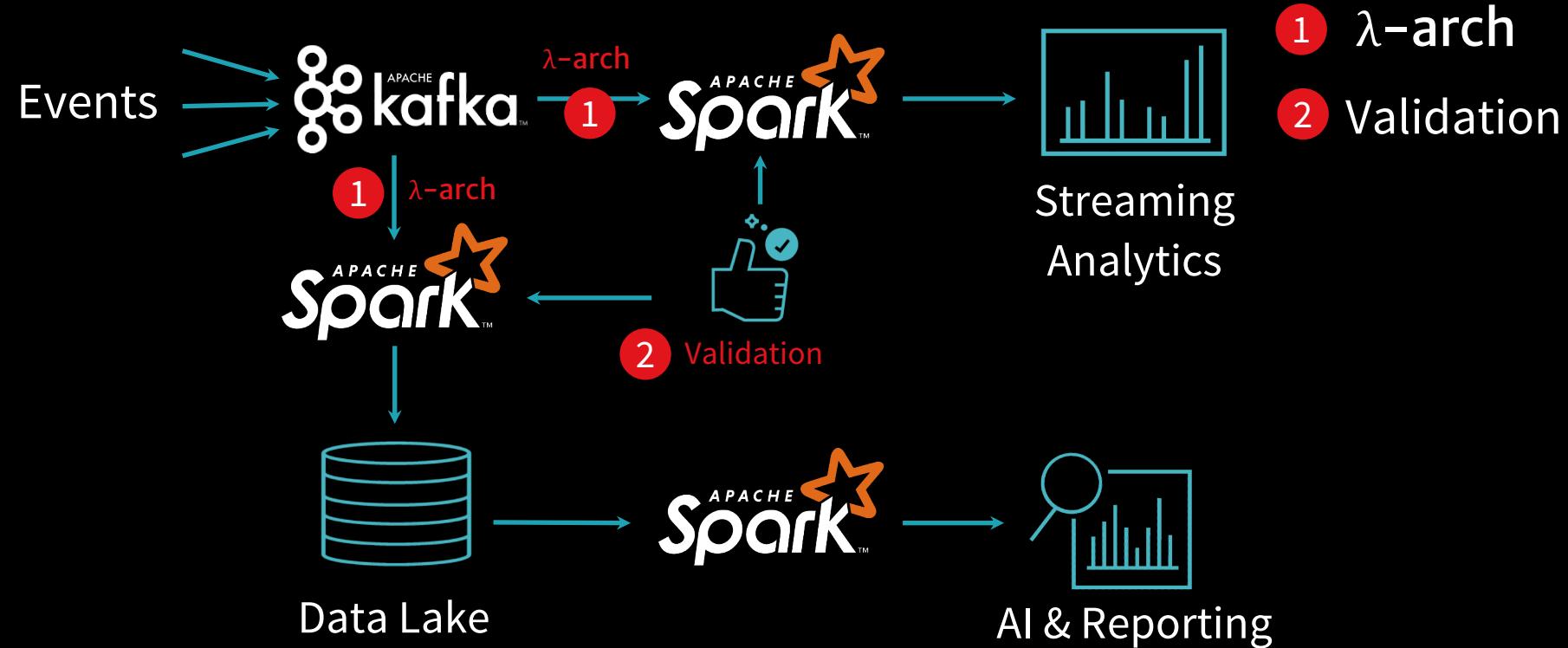
AI & Reporting



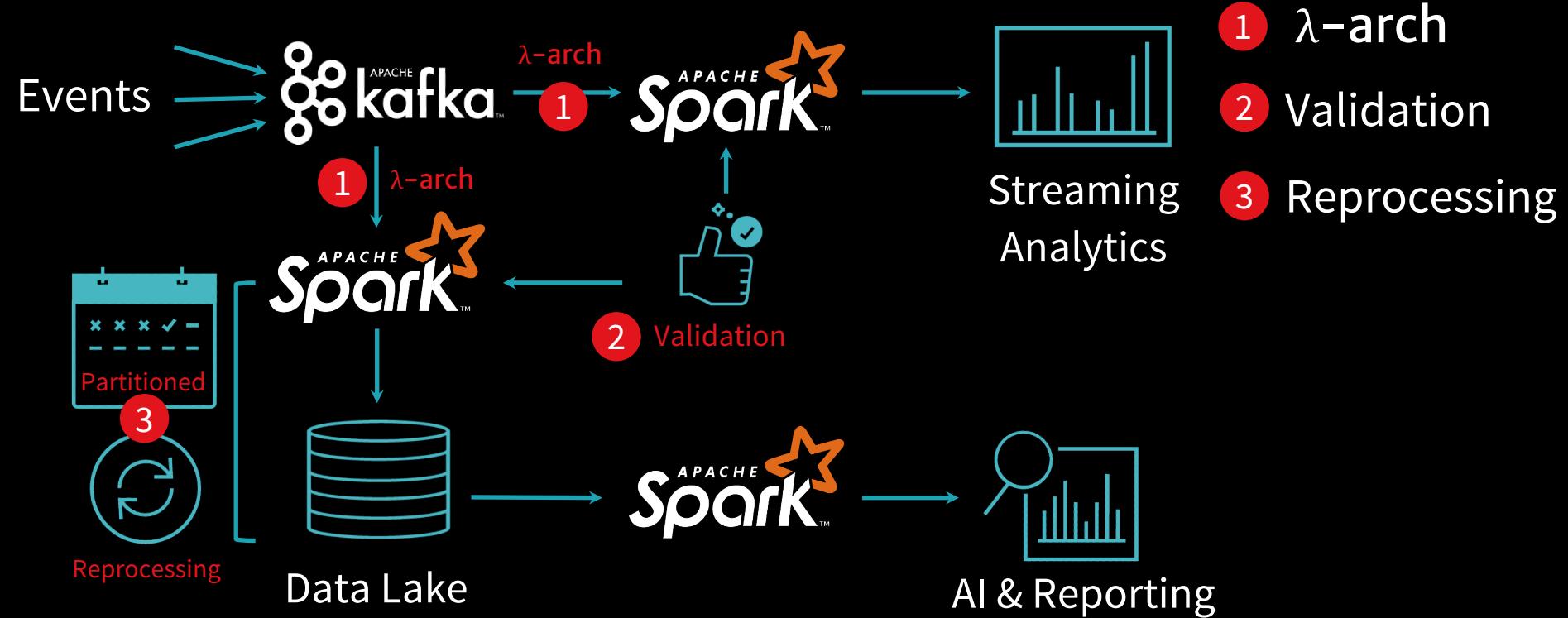
Challenge #1: Historical Queries?



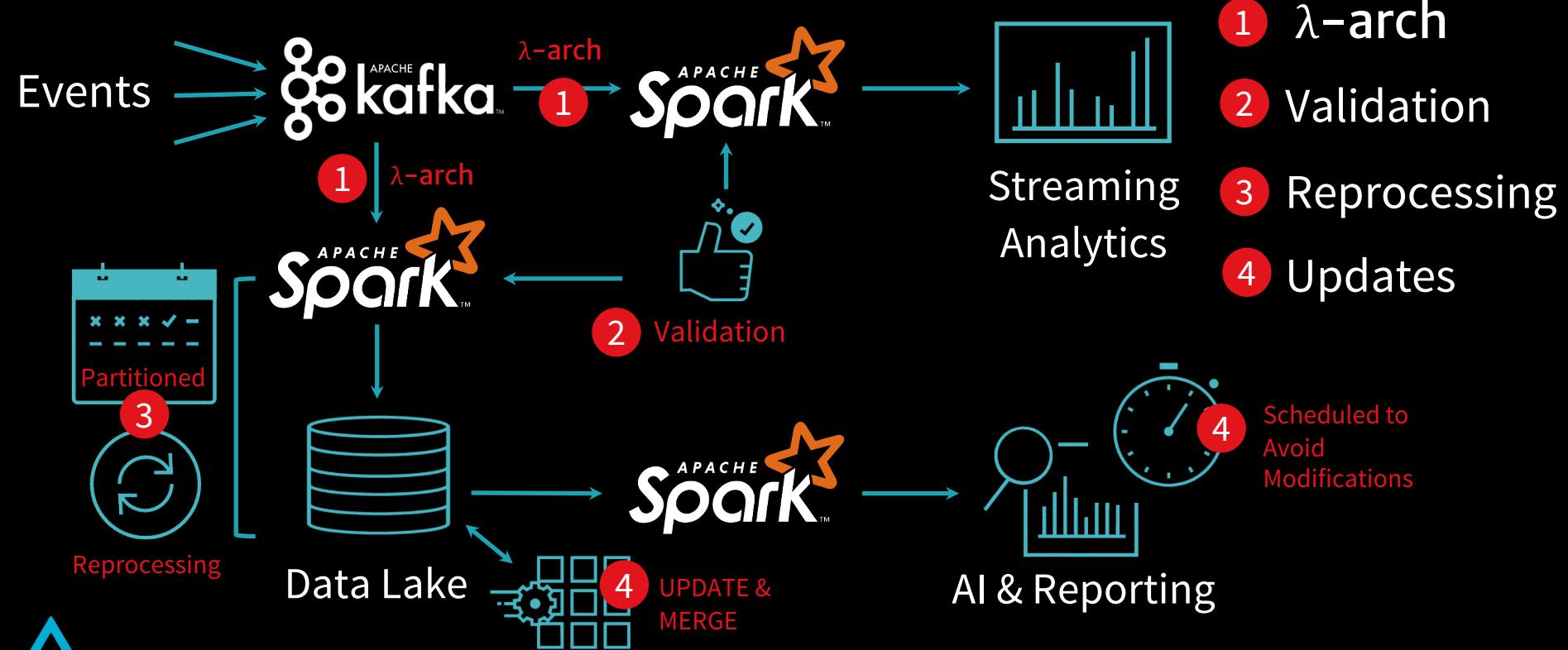
Challenge #2: Messy Data?



Challenge #3: Mistakes and Failures?



Challenge #4: Updates?



Wasting Time & Money

Solving Systems Problems

Instead of Extracting Value From Data



Data Lake Distractions



No atomicity means failed production jobs leave data in corrupt state requiring tedious recovery



No quality enforcement creates inconsistent and unusable data



No consistency / isolation makes it almost impossible to mix appends and reads, batch and streaming



OSS Delta Lake Key Features

Feature

ACID Transactions	Delta Lake brings ACID transactions to your data lakes. It provides serializability, the strongest level of isolation level. Learn more at Diving into Delta Lake: Unpacking the Transaction Log .
Scalable Metadata Handling	Delta Lake can handle petabyte-scale tables with billions of partitions and files at ease.
Time Travel (data versioning)	Delta Lake provides data snapshots to access and revert to earlier versions of data for audits, rollbacks or to reproduce experiments.
Open Format	All data in Delta Lake is stored in Apache Parquet format enabling Delta Lake to leverage the efficient compression and encoding schemes that are native to Parquet
Unified Batch and Streaming Source and Sink	A table in Delta Lake is both a batch table, as well as a streaming source and sink. Streaming data ingest, batch historic backfill, and interactive queries all just work out of the box.



OSS Delta Lake Key Features (Continued)

Feature

Schema Enforcement and Evolution

Delta Lake provides the ability to specify your schema and enforce it. This helps ensure that the data types are correct and required columns are present, preventing bad data from causing data corruption. For more information, refer to [Diving Into Delta Lake: Schema Enforcement & Evolution](#).

Audit History

Delta Lake transaction log records details about every change made to data providing a full audit trail of the changes.

DML Operations

Delta Lake supports SQL, [Scala / Java](#) and [Python](#) APIs to merge, update and delete datasets allowing you to easily comply with GDPR and CCPA and simplifying use cases like change data capture. For more information, refer to [Diving Into Delta Lake: DML Internals](#)

100% Compatible with Apache Spark API

Developers can use Delta Lake with their existing data pipelines with minimal change as it is fully compatible with Spark, the commonly used big data processing engine.



Delta Lake CY21H2 Roadmap

<https://github.com/delta-io/delta/issues/748>



- Delta Standalone Reader (Q3)
- Delta Standalone Writer (Q4)



- Flink/Delta Source (Q4)
- Flink/Delta Sync (22Q1)



- Pulsar/Delta Source (Q4)
- Pulsar/Delta Sync (22Q1)



- Spark 3.2 Support (Q4)



Hive3 Connector (Q3)



LakeFS Integration (Q3)

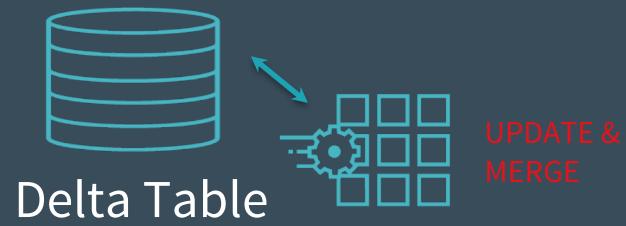


Nessie Integration (Q4)

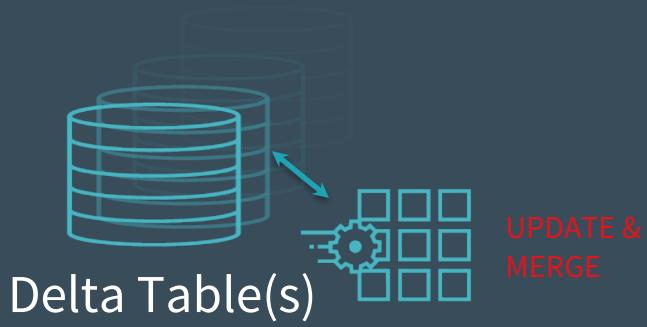


Community meetings: every two weeks have meetings with Trino, Flink, Nessie, LakeFS, Pulsar, Rust, and Core

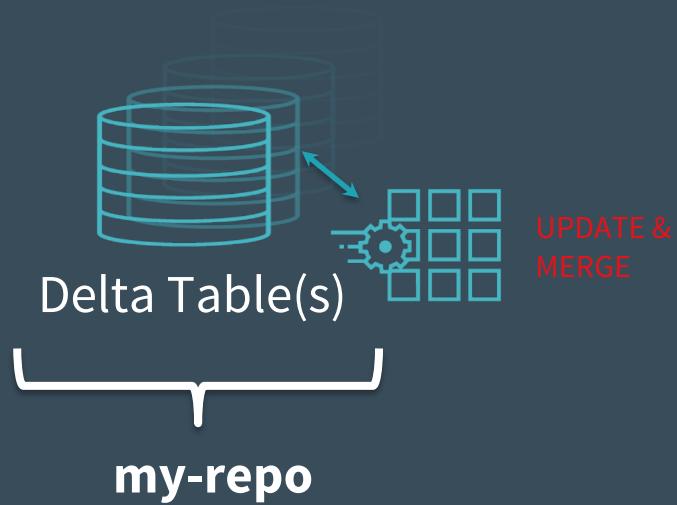
Single-Table Consistency



Multi - Table Consistency

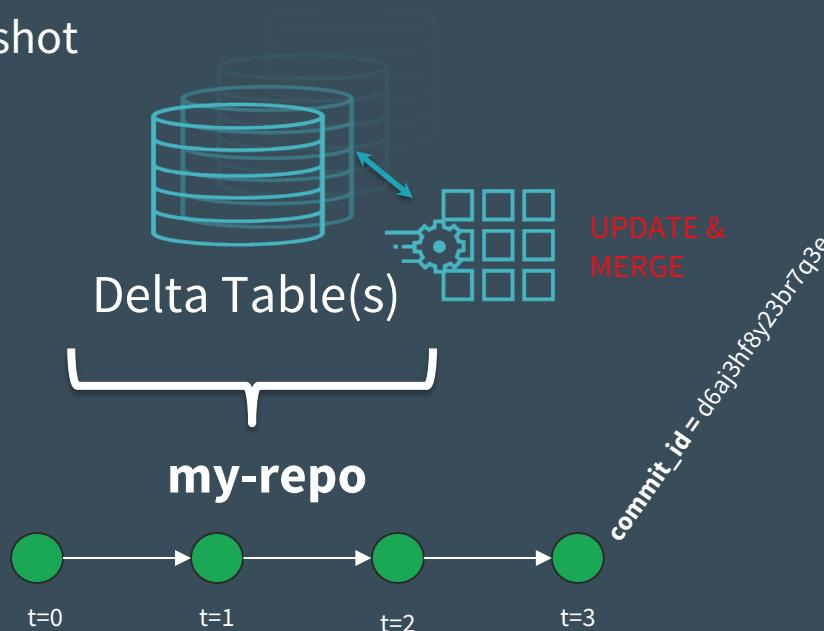


Multi - Table Consistency



Multi - Table Consistency

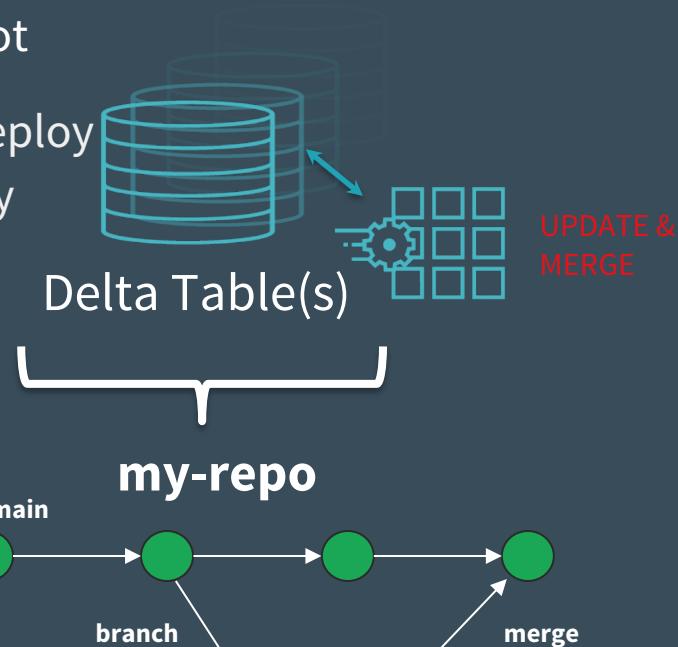
1. **commit** - save cross collection snapshot



Multi - Table Consistency

1. **commit** - save cross collection snapshot

2. **branch/merge** - deploy data atomically



How it works – Metadata All the Way Down



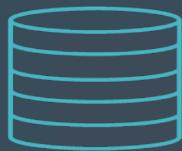
Transaction Log
Table Versions

(Optional) Partition Directories
Data Files

my_table/
 → _delta_log/
 → 00000.json
 → 00001.json
 → date=2019-01-01/
 → file-1.parquet



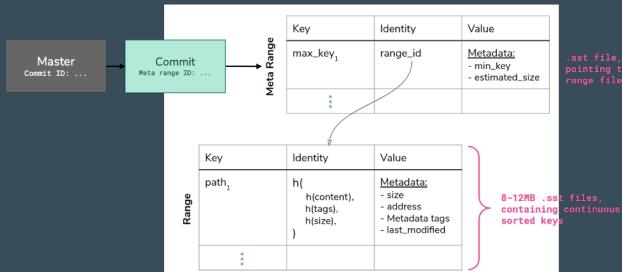
How it works – Metadata All the Way Down



Transaction Log
Table Versions

(Optional) Partition Directories
Data Files

my_table/
 →_delta_log/
 →0000.json
 →0001.json
 date=2019-01-01/
 ↓file-1.parquet



Object Store



Configuring lakeFS + Delta



Configuring lakeFS + Delta

1. Set Spark configs to S3 API endpoint

```
spark.hadoop.fs.s3a.bucket.<repo-name>.access.key AKIAIOSFODNN7EXAMPLE  
spark.hadoop.fs.s3a.bucket.<repo-name>.secret.key wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY  
spark.hadoop.fs.s3a.bucket.<repo-name>.endpoint https://lakefs.example.com  
spark.hadoop.fs.s3a.path.style.access true
```

Configuring lakeFS + Delta

1. Set Spark configs to S3 API endpoint

```
spark.hadoop.fs.s3a.bucket.<repo-name>.access.key AKIAIOSFODNN7EXAMPLE  
spark.hadoop.fs.s3a.bucket.<repo-name>.secret.key wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY  
spark.hadoop.fs.s3a.bucket.<repo-name>.endpoint https://lakefs.example.com  
spark.hadoop.fs.s3a.path.style.access true
```

2. Update object store location paths

```
Cmd 8  
1 %sql  
2 DROP TABLE IF EXISTS d_loan_states;  
3  
4 CREATE TABLE d_loan_states  
5 USING delta  
6 LOCATION 's3a://bi-reports/dev-reports/tables/loan_by_state'  
7 AS SELECT * FROM loan_by_state;
```



The diagram shows two labels, "repo" and "branch", positioned above the "LOCATION" and "AS SELECT" clauses respectively. Two arrows originate from these labels and point towards their respective corresponding clauses in the SQL code.

▶ (3) Spark Jobs

Recap

- Add your Delta tables to a lakeFS repository.
- Utilize git-inspired operations to provide atomicity on transactions across multiple tables.

Who is using  **DELTA LAKE**?



Used by 5000+ of organizations world wide

> 1+ exabyte processed every three days

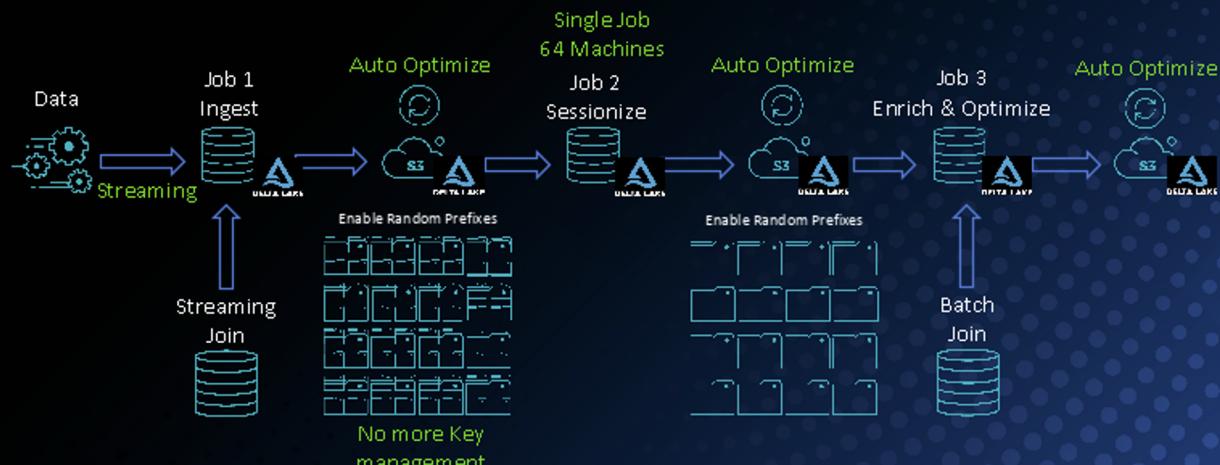


Barracuda





SESSIONIZATION WITH DELTA LAKE



FASTER QUERIES, RELIABLE PIPELINES, 10X REDUCTION IN COMPUTE!



Improved reliability:
Petabyte-scale jobs

10x lower compute:
640 instances to 64!

Simpler, faster ETL:
84 jobs → 3 jobs
halved data latency



How do I use  **DELTA LAKE** ?



Get Started with Delta using Spark APIs

Add Spark Package

```
pyspark --packages io.delta:delta-core_2.12:0.1.0  
bin/spark-shell --packages io.delta:delta-core_2.12:0.1.0
```

Maven

```
<dependency>  
  <groupId>io.delta</groupId>  
  <artifactId>delta-core_2.12</artifactId>  
  <version>0.1.0</version>  
</dependency>
```

Instead of **parquet**...

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

... simply say **delta**

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



In Progress: Declarative Pipelines



Enforce metadata, storage, and quality declaratively.

```
dataset("warehouse")
  .query(input("kafka").select(...).join(...))      // Query to materialize
  .location(...)                                     // Storage Location
  .schema(...)                                       // Optional strict schema checking
  .metastoreName(...)                                // Hive Metastore
  .description(...)                                  // Human readable description
  .expect("validTimestamp",
    "timestamp > 2012-01-01 AND ...",
    "fail / alert / quarantine")                     // Expectations on data quality
```

*Coming Soon



In Progress: Declarative Pipelines



Enforce metadata, storage, and quality declaratively.

```
dataset("warehouse")
  .query(input("kafka").withColumn(...).join(...)) // Query to materialize
  .location(...) // Storage Location
  .schema(...) // Optional strict schema checking
  .metastoreName(...) // Hive Metastore
  .description(...) // Human readable description
  .expect("validTimestamp",
    "timestamp > 2012-01-01 AND ...",
    "fail / alert / quarantine") // Expectations on data quality
```

*Coming Soon



In Progress: Declarative Pipelines



Enforce metadata, storage, and quality declaratively.

```
dataset("warehouse")
  .query(input("kafka").withColumn(...).join(...)) // Query to materialize
  .location(...) // Storage Location
  .schema(...) // Optional strict schema checking
  .metastoreName(...) // Hive Metastore
  .description(...) // Human readable description
  .expect("validTimestamp",
    "timestamp > 2012-01-01 AND ...",
    "fail / alert / quarantine") // Expectations on data quality
```

*Coming Soon

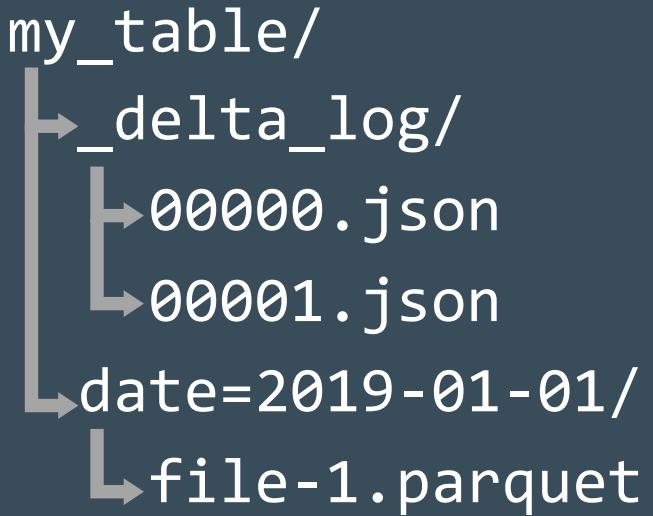


How does DELTA LAKE work?



Delta On Disk

Transaction Log
Table Versions
(Optional) Partition Directories
Data Files



Version N = Version N-1 + Actions

Change Metadata – name, schema, partitioning, etc

Add File – adds a file (with optional statistics)

Remove File – removes a file

Result: Current Metadata, List of Files



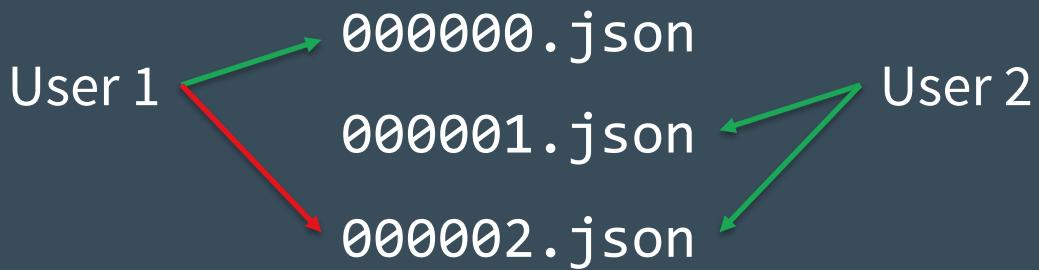
Implementing Atomicity

Changes to the table
are stored as
ordered, atomic
units called commits



Ensuring Serializability

Need to agree on the order of changes, even when there are multiple writers.



Solving Conflicts Optimistically

1. Record start version
2. Record reads/writes
3. Attempt commit
4. If someone else wins,
check if anything you
read has changed.
5. Try again.

Read: Schema

Write: Append

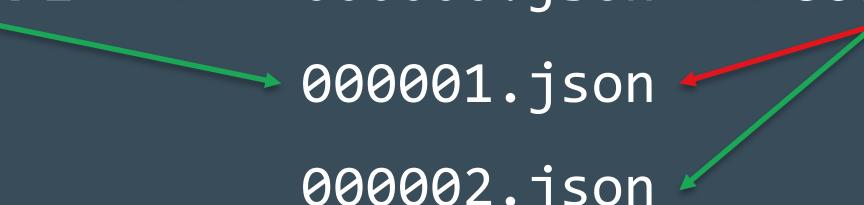
User 1



Read: Schema

Write: Append

User 2



Handling Massive Metadata

Large tables can have millions of files in them! How do we scale the metadata? Use Spark for scaling!

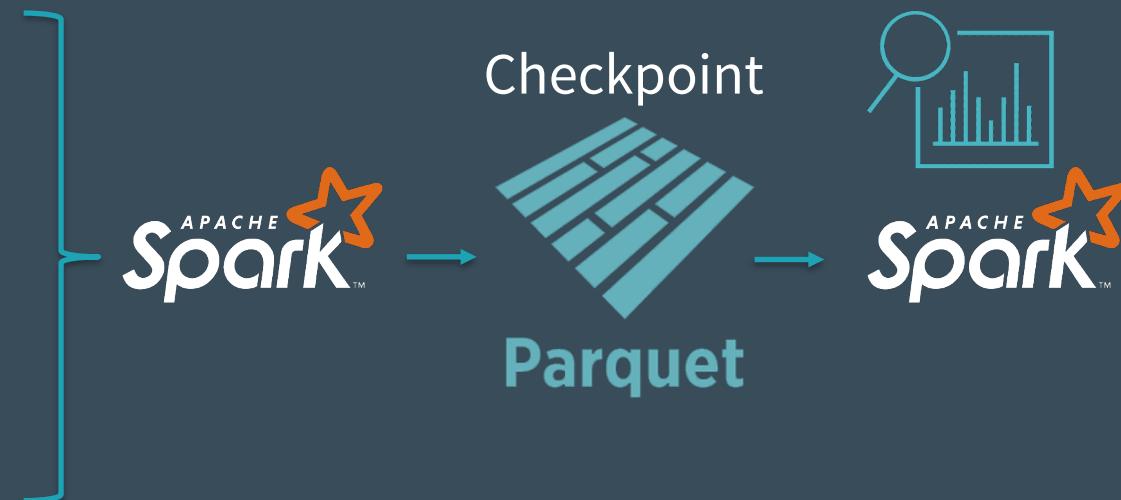
Add 1.parquet

Add 2.parquet

Remove 1.parquet

Remove 2.parquet

Add 3.parquet



Build your own Delta Lake
at **<https://delta.io>**

