What does a typical
data lake project look like?

Evolution of a Cutting-Edge Data Lake

Evolution of a Cutting-Edge Data Lake

Challenge #1: Historical Queries?
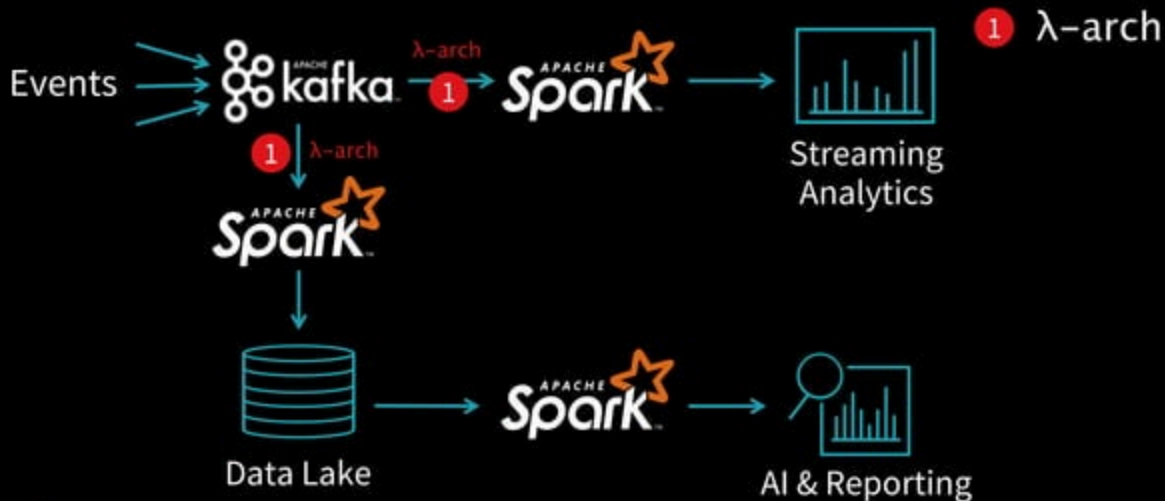
Challenge #2: Messy Data?
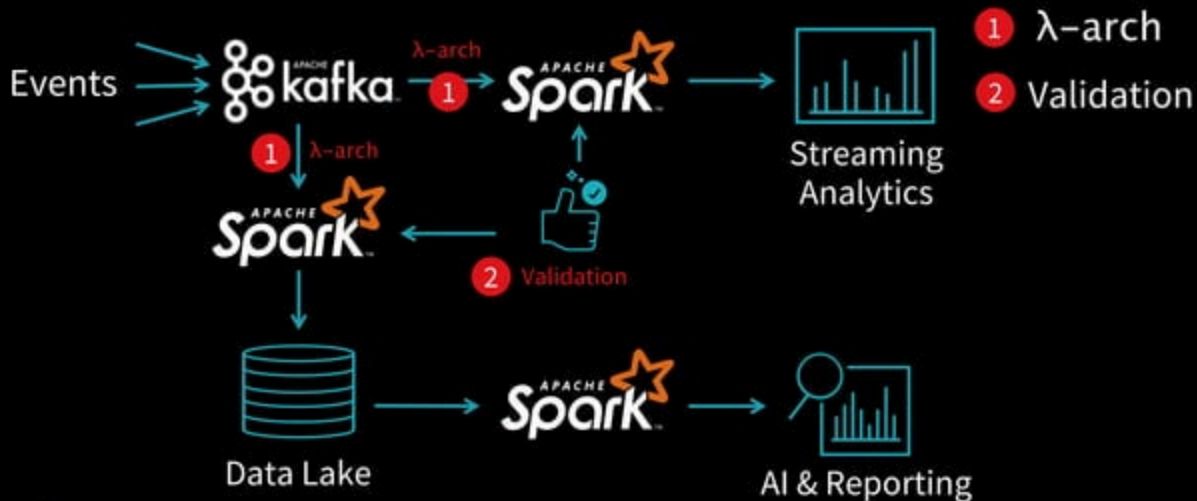
Challenge #3: Mistakes and Failures?

Challenge #4: Updates?

Events → **kafka** APACHE

λ-arch ① → **Spark** APACHE → Streaming Analytics

① λ-arch → **Spark** APACHE

Partitioned ③ Reprocessing → Data Lake 🛢 ← ② Validation 👍

Data Lake ⇄ ④ UPDATE & MERGE → **Spark** APACHE → AI & Reporting ④ Scheduled to Avoid Modifications
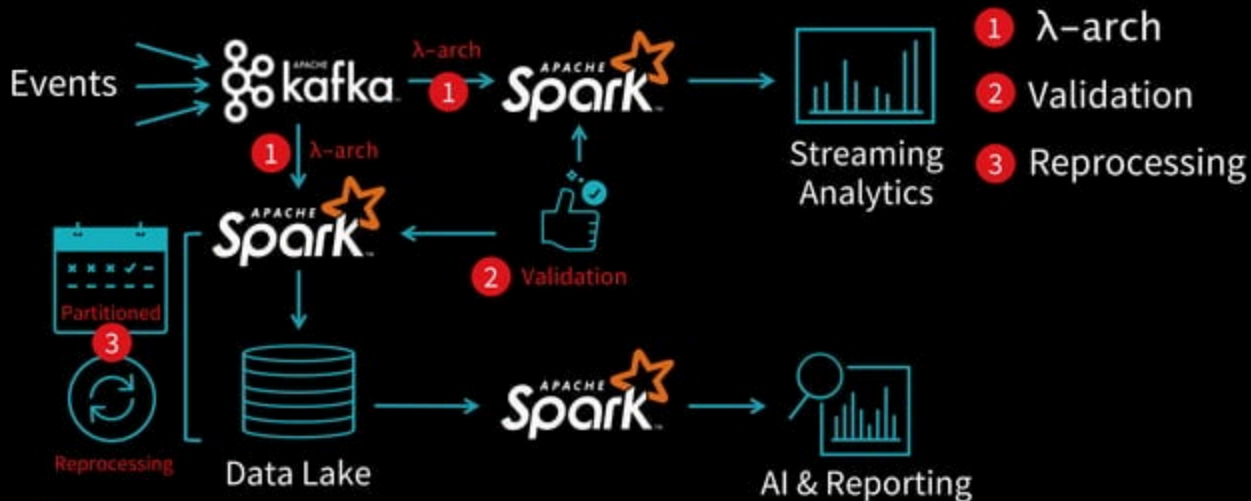
① λ-arch
② Validation
③ Reprocessing
④ Updates
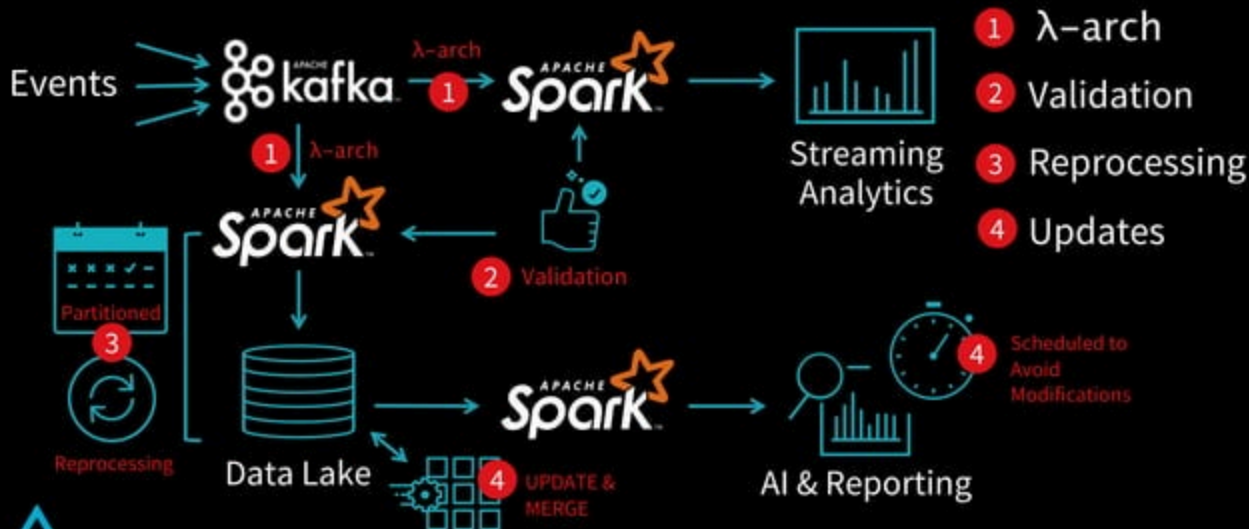
Wasting Time & Money

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

Challenges of the Data Lake

Events → Apache Kafka

λ-arch ① → Apache Spark → Streaming Analytics

① λ-arch ↓ Apache Spark

Partitioned ③ Reprocessing

② Validation

Data Lake → Apache Spark → AI & Reporting

④ UPDATE & MERGE

④ Scheduled to Avoid Modifications

1. λ-arch
2. Validation
3. Reprocessing
4. Updates

The ◭ DELTA LAKE Architecture

Streaming Analytics

AI & Reporting

Full ACID Transaction

Focus on your data flow, instead of worrying about failures.

The △ DELTA LAKE Architecture

Streaming Analytics

AI & Reporting

Open Standards, Open Source (Apache License)

Store petabytes of data without worries of lock-in. Growing community including Presto, Spark and more.

The Delta Lake Architecture

kafka → Kinesis → Data Lake (CSV, JSON, TXT...) → kafka

Streaming Analytics

AI & Reporting

Powered by Apache Spark

Unifies Streaming / Batch. Convert existing jobs with minimal modifications.

The Δ DELTA LAKE

Data Quality Levels

	Bronze	Silver	Gold	
kafka				Streaming Analytics
Kinesis				
CSV, JSON, TXT. Data Lake	Raw Ingestion	Filtered, Cleaned Augmented	Business-level Aggregates	AI & Reporting
Spark				

Quality →

Delta Lake allows you to *incrementally* improve the quality of your data until it is ready for consumption.

The Delta Lake

Kafka, Kinesis, Data Lake (CSV, JSON, TXT.), Spark → Bronze (Raw Ingestion) → Silver (Filtered, Cleaned Augmented) → Gold (Business-level Aggregates) → Streaming Analytics, AI & Reporting

- Dumping ground for raw data
- Often with long retention (years)
- Avoid error-prone parsing

The Delta Lake

Kafka, Kinesis, Data Lake (CSV, JSON, TXT.), Spark → Bronze (Raw Ingestion) → Silver (Filtered, Cleaned Augmented) → Gold (Business-level Aggregates) → Streaming Analytics, AI & Reporting

Clean data, ready for consumption.
Read with Spark or Presto*

*Coming Soon

The ◭ DELTA LAKE

Kafka → **Kinesis** → **CSV, JSON, TXT. Data Lake** → **Spark**

Bronze — Raw Ingestion

Silver — Filtered, Cleaned Augmented

Gold — Business-level Aggregates

→ **Streaming Analytics**

→ **AI & Reporting**
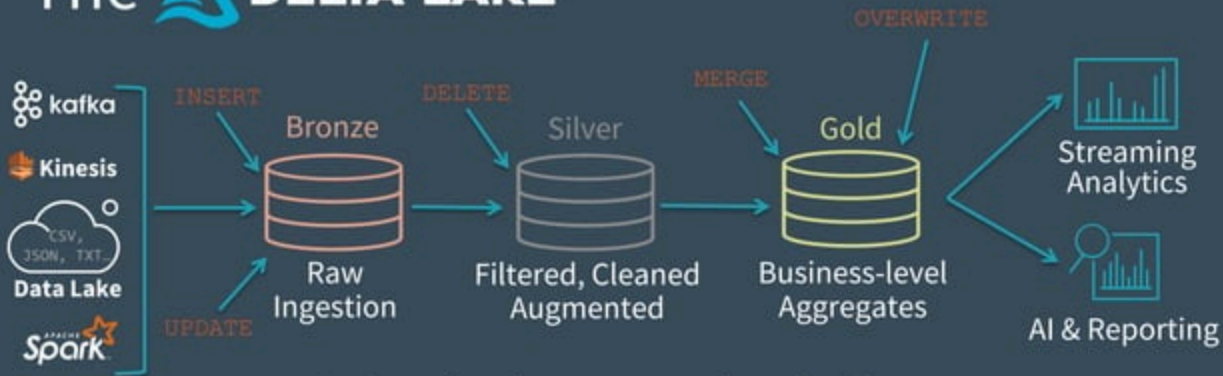
Streams move data through the Delta Lake
- Low-latency or manually triggered
- Eliminates management of schedules and jobs

The Delta Lake

Kafka, Kinesis, CSV/JSON/TXT Data Lake, Spark →
Bronze — Raw Ingestion →
DELETE Silver — Filtered, Cleaned Augmented →
DELETE Gold — Business-level Aggregates →
Streaming Analytics
AI & Reporting

Easy to recompute when business logic changes:

- Clear tables
- Restart streams

Who is using △ DELTA LAKE?

Used by 1000s of organizations world wide

> 1 exabyte processed last month alone

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

```xml
<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")
```

Data Quality

Enforce metadata, storage, and quality declaratively.

```
table("warehouse")
  .location(…)              // Location on DBFS
  .schema(…)                // Optional strict schema checking
  .metastoreName(…)         // Registration in Hive Metastore
  .description(…)           // Human readable description for users
  .expect("validTimestamp", // Expectations on data quality
    "timestamp > 2012-01-01 AND …",
    "fail / alert / quarantine")
```

*Coming Soon

Data Quality

Enforce metadata, storage, and quality declaratively.

```
table("warehouse")
  .location(…)                    // Location on DBFS
  .schema(…)                      // Optional strict schema checking
  .metastoreName(…)               // Registration in Hive Metastore
  .description(…)                 // Human readable description for users
  .expect("validTimestamp",       // Expectations on data quality
    "timestamp > 2012-01-01 AND …",
    "fail / alert / quarantine")
```

*Coming Soon

How does **DELTA LAKE** work?

Delta On Disk

Table = result of a set of actions

Change Metadata – name, schema, partitioning, etc
Add File – adds a file (with optional statistics)
Remove File – removes a file

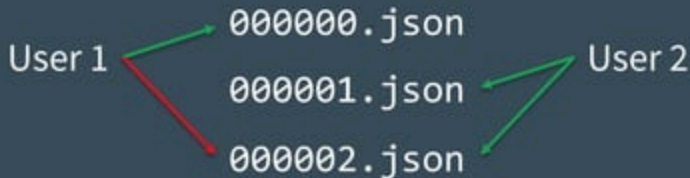Result: Current Metadata, List of Files, List of Txns, Version

Implementing Atomicity

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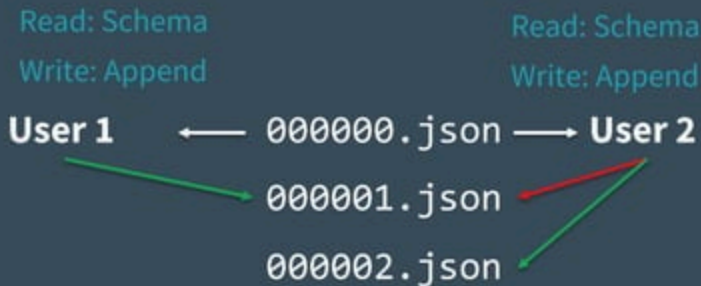*ordered, atomic*
units called commits

000000.json

000001.json

...

Add 1.parquet

Add 2.parquet

Remove 1.parquet

Remove 2.parquet

Add 3.parquet

Ensuring Serializablity

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

000000.json

000001.json

000002.json

User 1

User 2

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 ⟵ `000000.json` ⟶ **User 2**

`000001.json`

`000002.json`

Read: Schema
Write: Append

Handling Massive Metadata

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

Road Map

- 0.2.0 – Released!
  - S3 Support
  - Azure Blob Store and ADLS Support
- 0.3.0 (~July)
  - UPDATE (Scala)
  - DELETE (Scala)
  - MERGE (Scala)
  - VACUUM (Scala)

- Rest of Q3
  - DDL Support / Hive Metastore
  - SQL DML Support