The Promise of the Data Lake

1. Collect Everything
   - Customer Data
   - Video/Speech
   - Emails/Web Pages
   - Sensor Data (IoT)
   - Click Streams

2. Store it all in the Data Lake

3. Data Science & Machine Learning
   - Recommendation Engines
   - Risk, Fraud Detection
   - IoT & Predictive Maintenance
   - Genomics & DNA Sequencing

Garbage In ➔ Garbage Stored ➔ Garbage Out
What does a typical **data lake** project look like?
Evolution of a Cutting-Edge Data Lake

Events ➔ Kafka

Data Lake

Streaming Analytics

AI & Reporting
Evolution of a Cutting-Edge Data Lake

Events → Apache Kafka → Apache Spark → Streaming Analytics

Data Lake

AI & Reporting
Challenge #1: Historical Queries?

Events → Apache Kafka → λ-arch → Apache Spark → Streaming Analytics

Data Lake → Apache Spark → λ-arch → AI & Reporting
Challenge #2: Messy Data?

Events → Kafka → Spark (λ-arch) → Streaming Analytics

1. λ-arch
2. Validation

Data Lake → Spark → Validation → AI & Reporting
Challenge #3: Mistakes and Failures?

1. λ-arch
2. Validation
3. Reprocessing
Challenge #4: Updates?

- **Events** → **Apache Kafka** → **Spark** → **Stream Analytics**
  - 1. λ-arch
  - 2. Validation
  - 3. Reprocessing
  - 4. Updates

- **Data Lake**
  - Partitioned
  - Reprocessing

- **AI & Reporting**
  - Scheduled to Avoid Modifications
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.
Let’s try it instead with DELTA LAKE
Challenges of the Data Lake

Events → Apache Kafka → Lambda Architecture → Apache Spark → Streaming Analytics

1. Lambda Architecture
2. Validation
3. Reprocessing
4. Updates

Data Lake → Apache Spark → Partitioned → Reprocessing

- Scheduled to Avoid Modifications
- Update & Merge
Delta Lake allows you to incrementally improve the quality of your data until it is ready for consumption.
The DELTA LAKE

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

- Raw Ingestion
- Filtered, Cleaned, Augmented
- Business-level Aggregates

- Kafka
- Kinesis
- CSV, JSON, TXT...

- Streaming Analytics
- AI & Reporting
The Delta Lake

Data Ingestion
- Raw
- Filtered, Cleaned, Augmented
- Business-level Aggregates

Intermediate data with some cleanup applied.
Queryable for easy debugging!
Clean data, ready for consumption.
Read with Spark or Presto*

*Coming Soon
Streams move data through the Delta Lake
- Low-latency or manually triggered
- Eliminates management of schedules and jobs
Delta Lake also supports batch jobs and standard DML

- Retention
- Corrections
- GDPR
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
Improved reliability:
Petabyte-scale jobs

10x lower compute:
640 instances to 64!

Simpler, faster ETL:
84 jobs → 3 jobs
halved data latency
Easier transactional updates:
No downtime or consistency issues!

Simple CDC:
Easy with MERGE

Improved performance:
Queries run faster
>1 hr → < 6 sec
How do I use Delta Lake?
Get Started with Delta using Spark APIs

Add Spark Package

```bash
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`...

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

... simply say `delta`

```python
dataframe
  .write
  .format("delta")
  .save("/data")
```
How does Delta Lake work?
Delta On Disk

- Transaction Log
- Table Versions
- (Optional) Partition Directories
- Data Files

my_table/

_delta_log/

00000.json
00001.json

date=2019-01-01/

file-1.parquet
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.

- Add 1.parquet
- Add 2.parquet
- Remove 1.parquet
- Remove 2.parquet
- Add 3.parquet
- 000000.json
- 000001.json
- ...


Ensuring Serializability

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

User 1
- 000000.json
- 000001.json
- 000002.json

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.

User 1  ➔ User 2
Read: Schema
Write: Append

User 1
Read: Schema
Write: Append

000000.json ➔ 000001.json ➔ 000002.json
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
Road Map

• 0.2.0 – Released!
  • S3 Support
  • Azure Blob Store and ADLS Support

• 0.3.0 Released!
  • UPDATE (Scala)
  • DELETE (Scala)
  • MERGE (Scala)
  • VACUUM (Scala)

• Rest of Q3
  • DDL Support / Hive Metastore
Build your own Delta Lake at https://delta.io