Keeping your Data Lake optimized is **HARD**

Especially if you need to support **data mutability**

But it doesn’t have to be!

DATA+AI SUMMIT 2022

The Road to a Robust Data Lake

Utilizing Delta Lake and Databricks to Map 150 Million Miles of Roads a Month

Itai Yaffe

Senior Solutions Architect, Databricks

Ofir Kerker

Data Platform Tech Lead, Nexar

ORGANIZED BY databricks

DATA+AI SUMMIT 2022

The Road to a Robust Data Lake

Utilizing Delta Lake and Databricks to Map ~~150~~ 180 Million Miles of Roads a Month

Itai Yaffe
Senior Solutions Architect, Databricks

Ofir Kerker
Data Platform Tech Lead, Nexar

ORGANIZED BY databricks

Introduction

Ofir Kerker

- Data Platform Tech Lead @ Nexar
- Former co-Founder & CTO @ Kapai
- Focusing on scalability and data engineering
- Ofir Kerker @ofirski_

Itai Yaffe

- Senior Solutions Architect @ Databricks
- Prev. Principal Solutions Architect @ Imply
- Prev. Big Data Tech Lead @ Nielsen
- Dealing with Big Data challenges since 2012
- Itai Yaffe @ItaiYaffe

What Will You Learn?

Efficiently process data in a streaming fashion,

What Will You Learn?

Efficiently process data in a streaming fashion,

support data mutability and keep your Data Lake optimized,

What Will You Learn?

Efficiently process data in a streaming fashion,

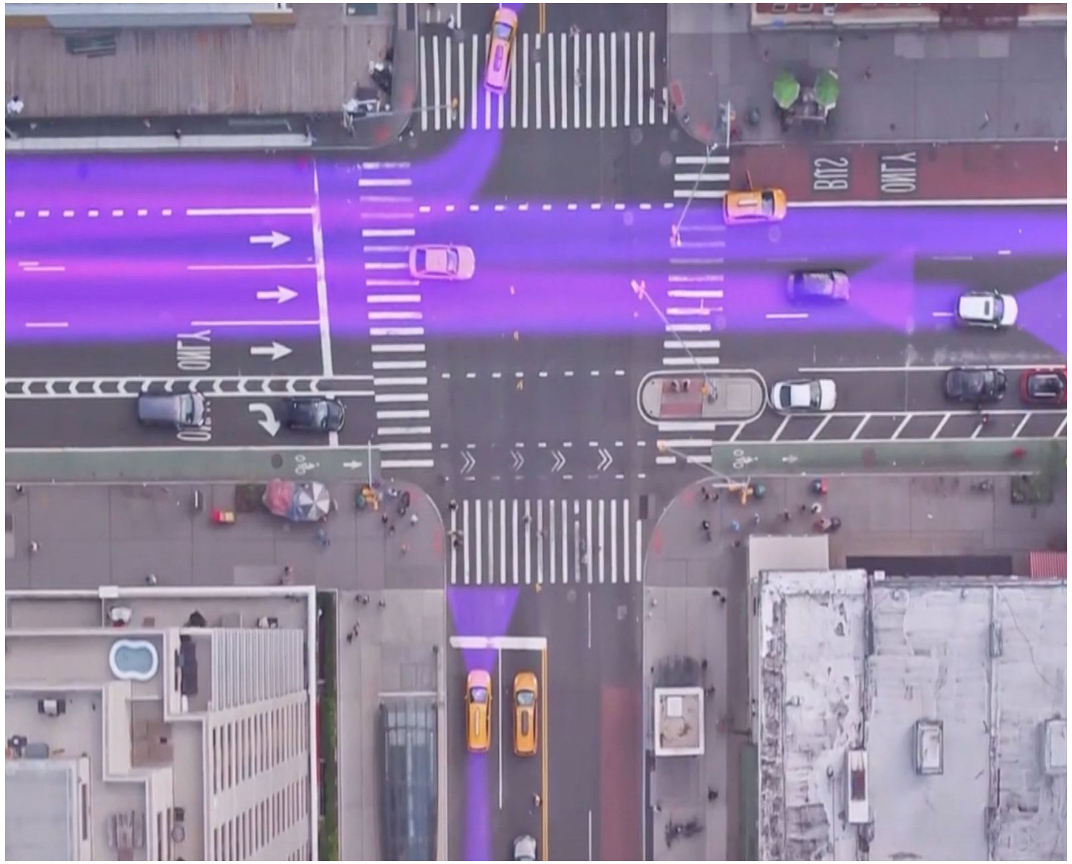
support data mutability and keep your Data Lake optimized,

by utilizing Delta Lake and Databricks

About Nexar

Nexar was founded with a moonshot mission of building the "air-traffic control" of the road

Nexar turns cars into vision sensors to build the first Digital Twin of the physical world



Nexar by the Numbers

COMPANY

130+
Employees in
TLV, NYC, Tokyo

+280%
YoY Growth

15%
US market share
in dash cams units

NETWORK

430K+
Nexar-powered
dash-cams on the road

180M
Video miles/month

4T
images collected
up to today

DASH-CAM

Cloud sync | Live streaming | Emergency access | Parking mode alerts | Road & cabin-facing ADAS | Parking spot detection

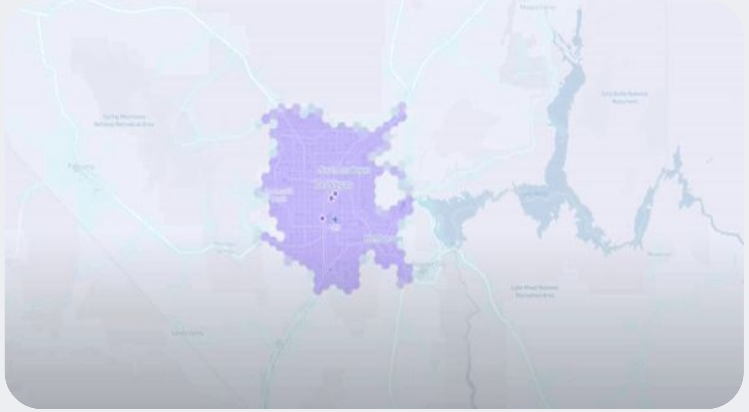
CUSTOMERS

Autonomous Vehicles | Public | Mapping providers | Fleets | Automotive OEMs | Insurance

Crowdsourced Vision

1

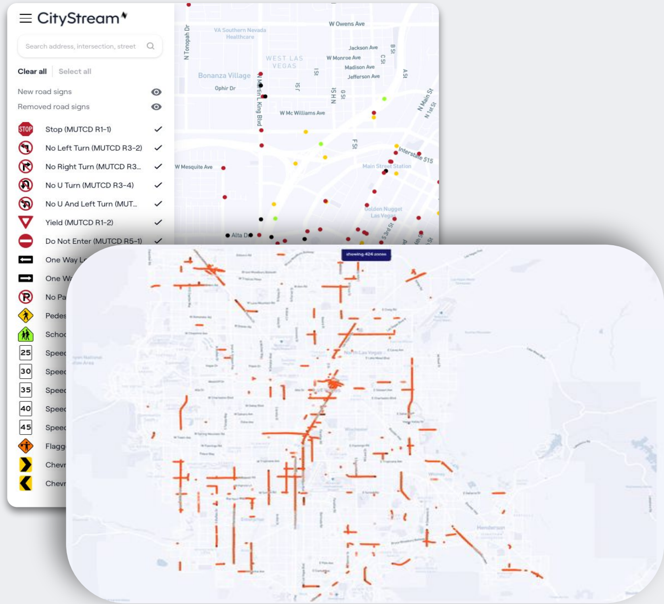
Imagery



Fresh and recent images and videos

2

AI-Based Detections



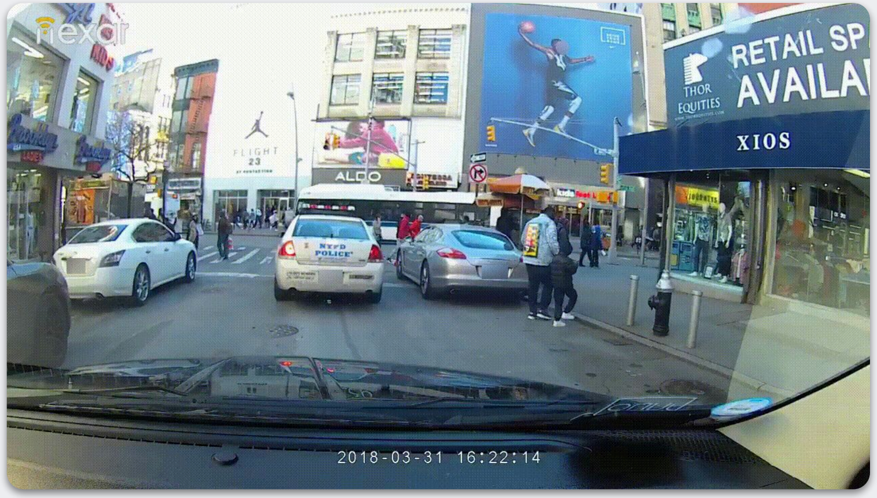
Road Signs, Traffic Lights, Road Work Zones, Potholes and Lane Markings

3

AI-Based Change Detection

Detect changes: fallen signs, scale-up enforcement and prioritize road maintenance

Privacy by Design

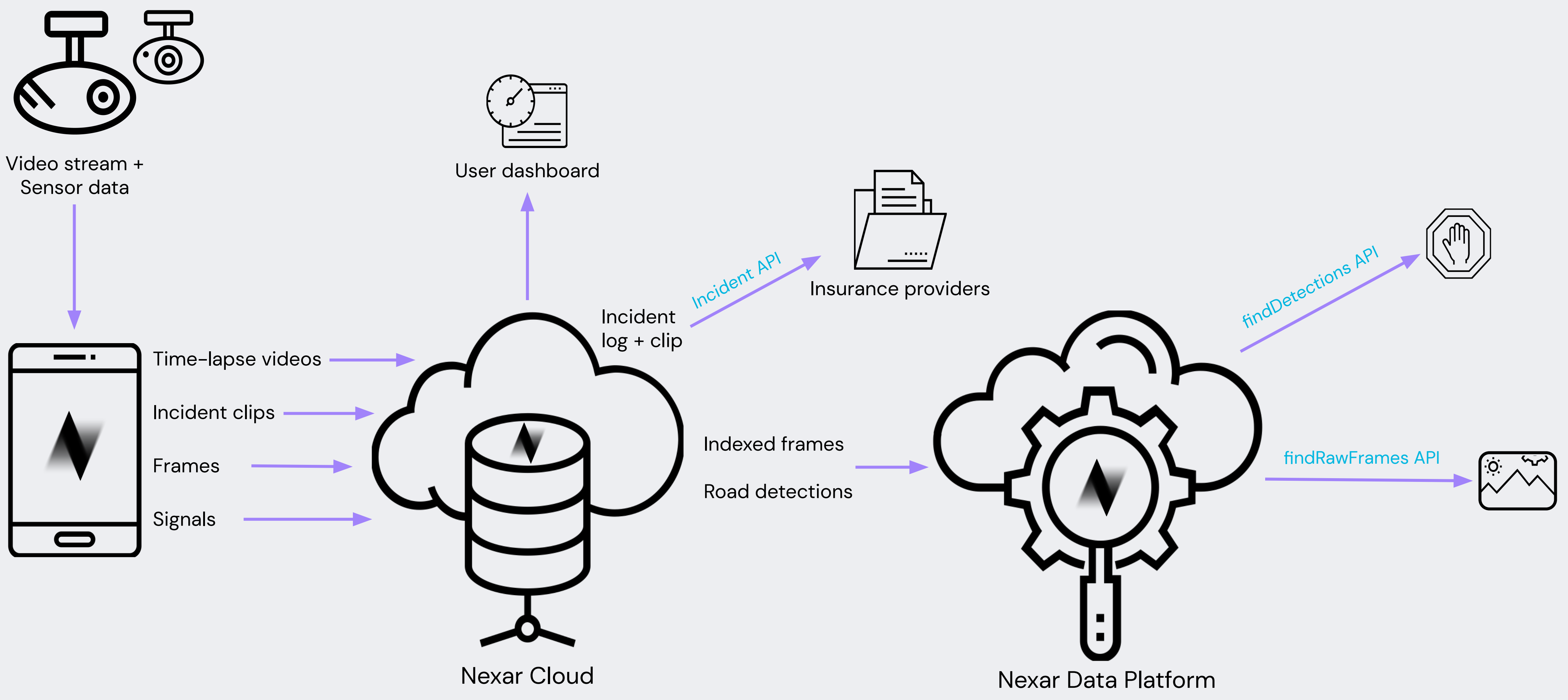


Using AI to remove all potential Personal Identifiable Information (PII)

- Automatic blurring of license plates, faces around the car
- Automatic cropping of images to avoid PII exposure of driver dashboard
- Doesn’t share routes, personal footage
- GDPR and CCPA compliant

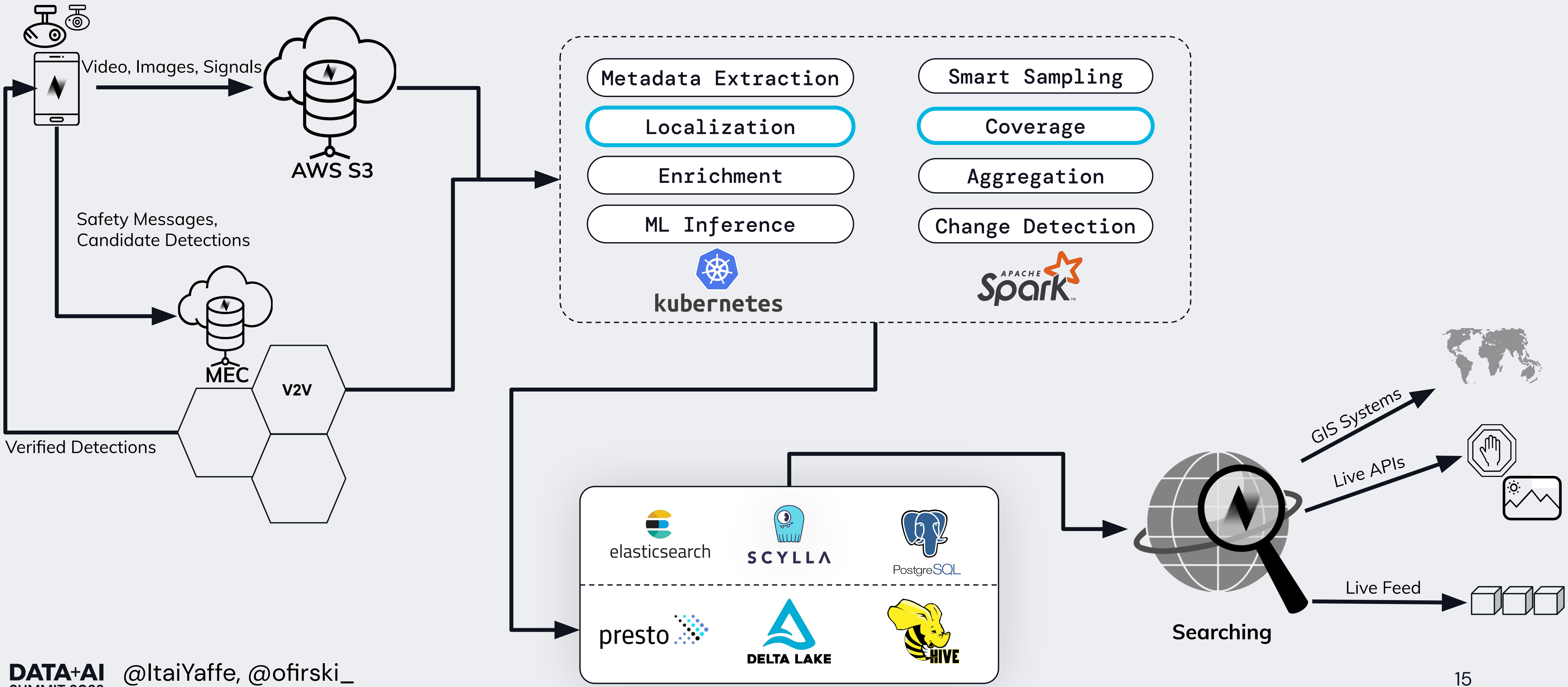
Nexar Platform High-Level Architecture

From edge devices to insights



Nexar Data Platform Architecture

Diving into the platform

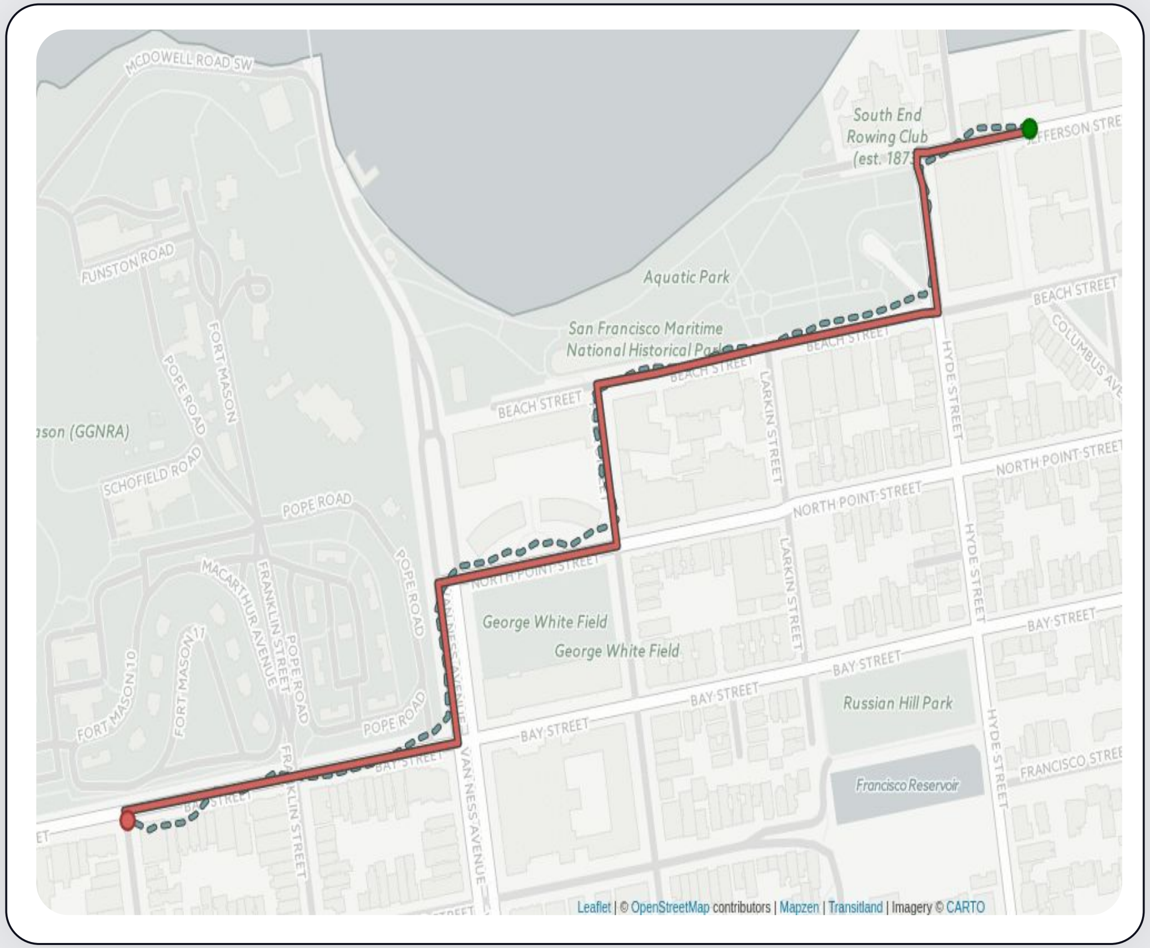


Ingesting a Stream of Files

- During every ride, the Nexar app collects signals into compressed files
- These files are uploaded directly to AWS S3 into a specific ride's prefix
- The data uploaded build up a massive amount of **small files**
- Each signals file independently goes through a pipeline of transformations and enrichments

The Small Files Challenge

- Transforming data in small chunks is straightforward
- However, querying data when it spans millions of small files using any SQL engine **isn't practical**:
  - Queries can run extremely slowly
  - Cost of compute becomes excessively high



Stream Processing Challenges

- In order to keep our Data Lake optimized, we need to compact **a stream of small files**
- Other challenges with stream processing include:
    - **Continuously updating** the imagery coverage across roads
    - Aggregate all previous day's events (from the Data Lake), taking into account **late-arriving data**

Let's Go Down Memory Lane…

Stream Processing in Traditional Data Lakes

A look back to 2019

- At Spark+AI Summit 2019 Europe, I shared the journey of building Nielsen Marketing Cloud's proprietary data infrastructure to mitigate some of the aforementioned challenges (see tinyurl.com/4s79mdpm)

Stream Processing in Traditional Data Lakes

A look back to 2019

- At Spark+AI Summit 2019 Europe, I shared the journey of building Nielsen Marketing Cloud's proprietary data infrastructure to mitigate some of the aforementioned challenges (see tinyurl.com/4s79mdpm)
- Nielsen is a data and measurement company
  - Nielsen Marketing Cloud - a unit within Nielsen
    - Anonymous device-level data is collected from various sources
    - The data (>5PB in total) is used for measurement and targeting

3 Methods of Data Processing

A look back to 2019

- Stream processing **into** a Data Lake

3 Methods of Data Processing

A look back to 2019

- Stream processing **into** a Data Lake
- Batch processing **from** a Data Lake

3 Methods of Data Processing

A look back to 2019

- Stream processing **into** a Data Lake
- Batch processing **from** a Data Lake
- Stream processing **over** a Data Lake

Stream Processing Into a Data Lake

What does it mean?

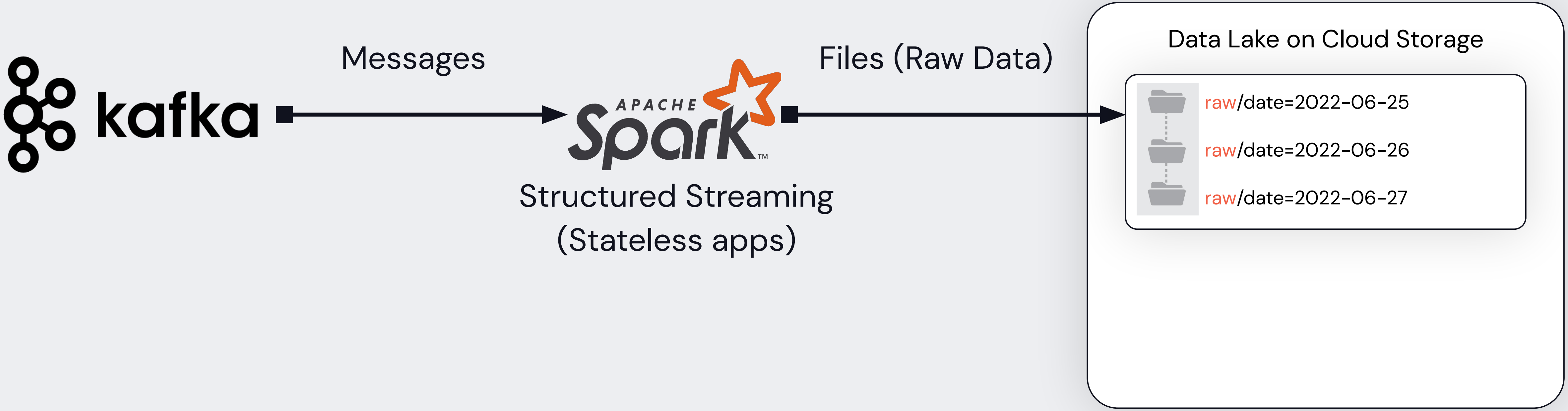
kafka → Messages → Apache Spark Structured Streaming → Files → Data Lake on Cloud Storage

Data Lake on Cloud Storage

- aws
- Microsoft Azure
- Google Cloud

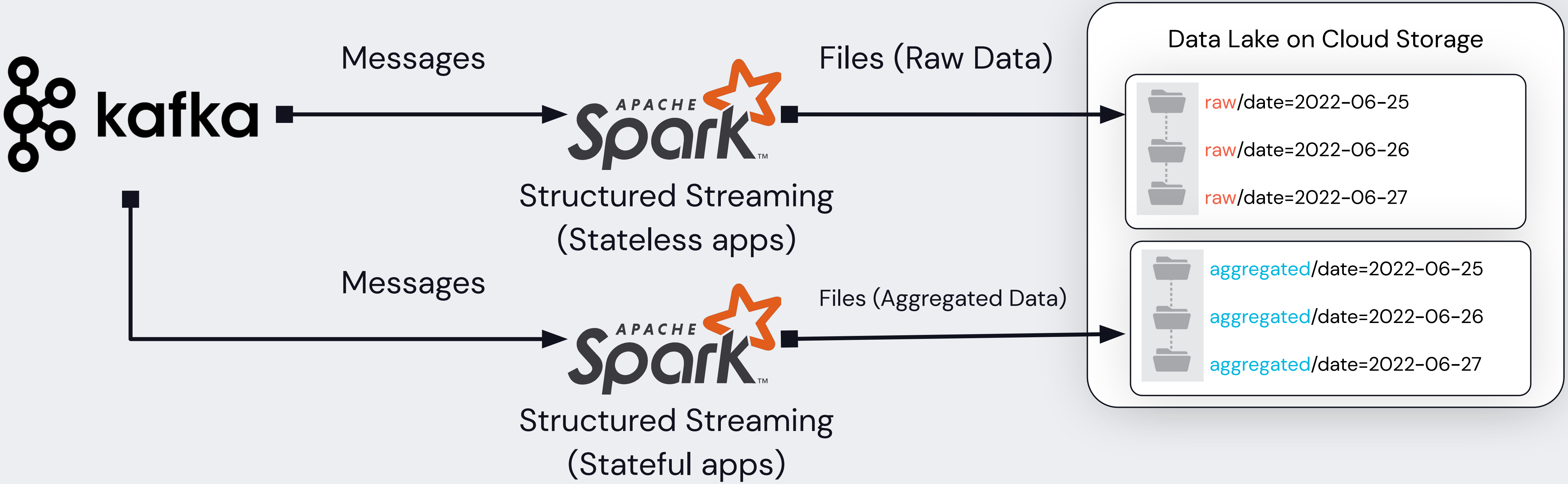
Stream Processing Into a Data Lake

What does it mean?



Stream Processing Into a Data Lake

What does it mean?



Stream Processing Into a Data Lake

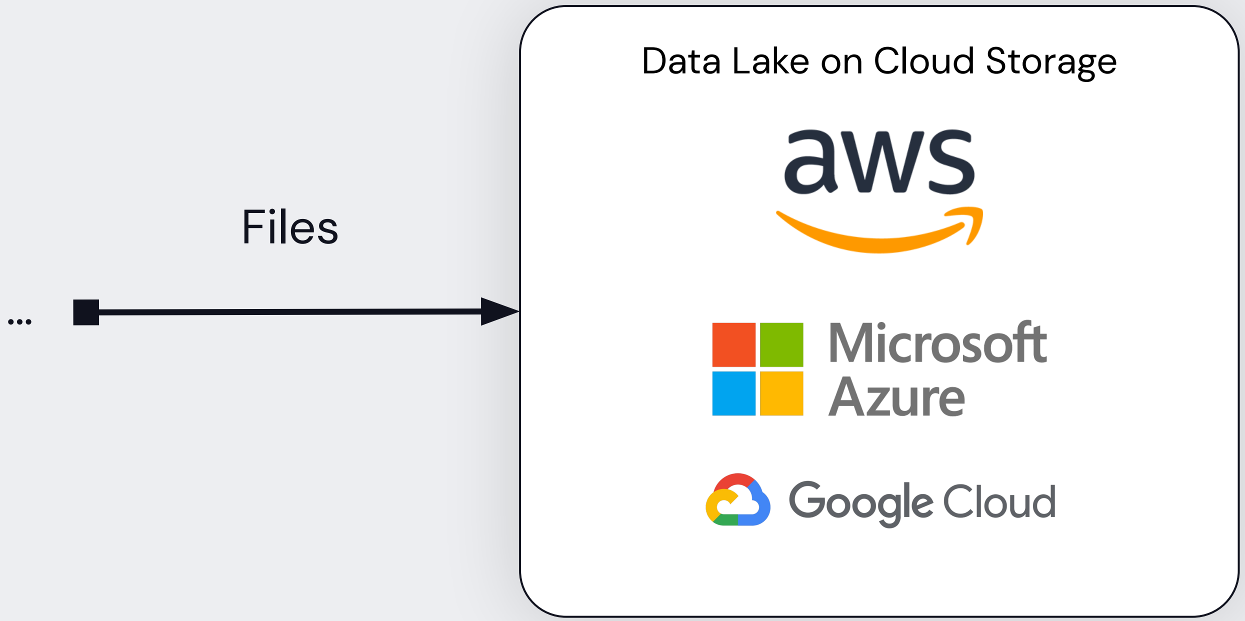
- Allows you to serve **fresher** data and enables **more informed business decisions**

Stream Processing Into a Data Lake

- Allows you to serve **fresher** data and enables **more informed business decisions**
- The additional (stateful) consumers **increase operational costs**
  - By consuming **write-optimized** data (e.g Avro messages) from message buses
  - By putting additional burden on the source system (e.g Kafka brokers)
  - By using **24/7** Streaming job clusters

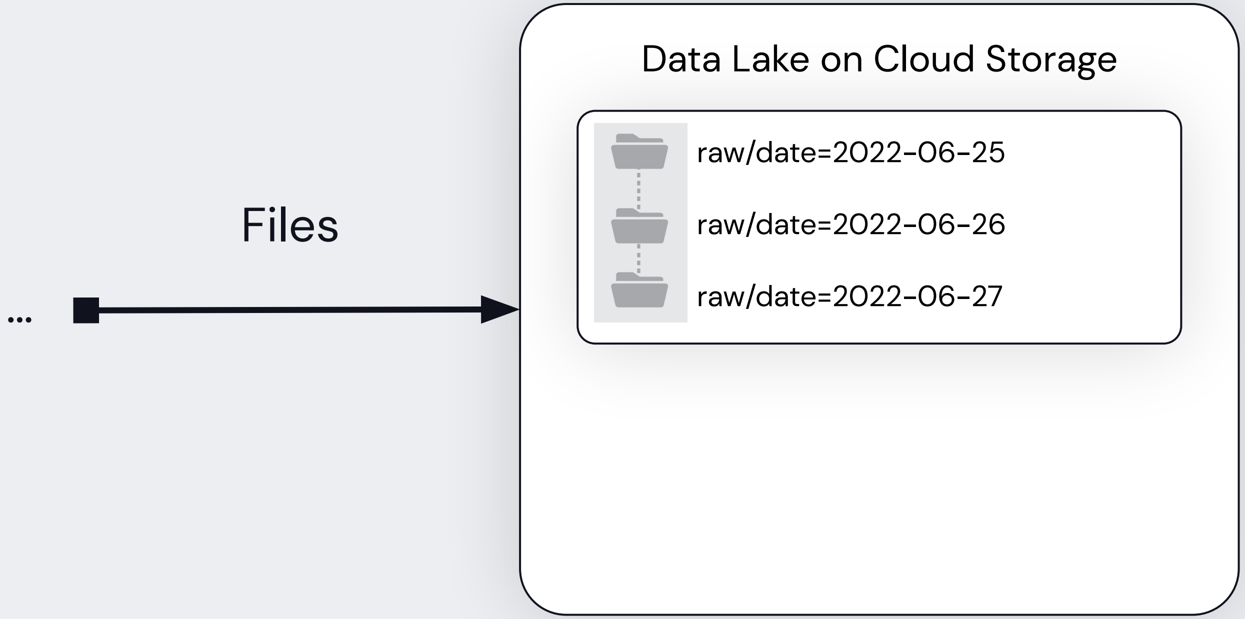
Batch Processing From a Data Lake

What does it mean?



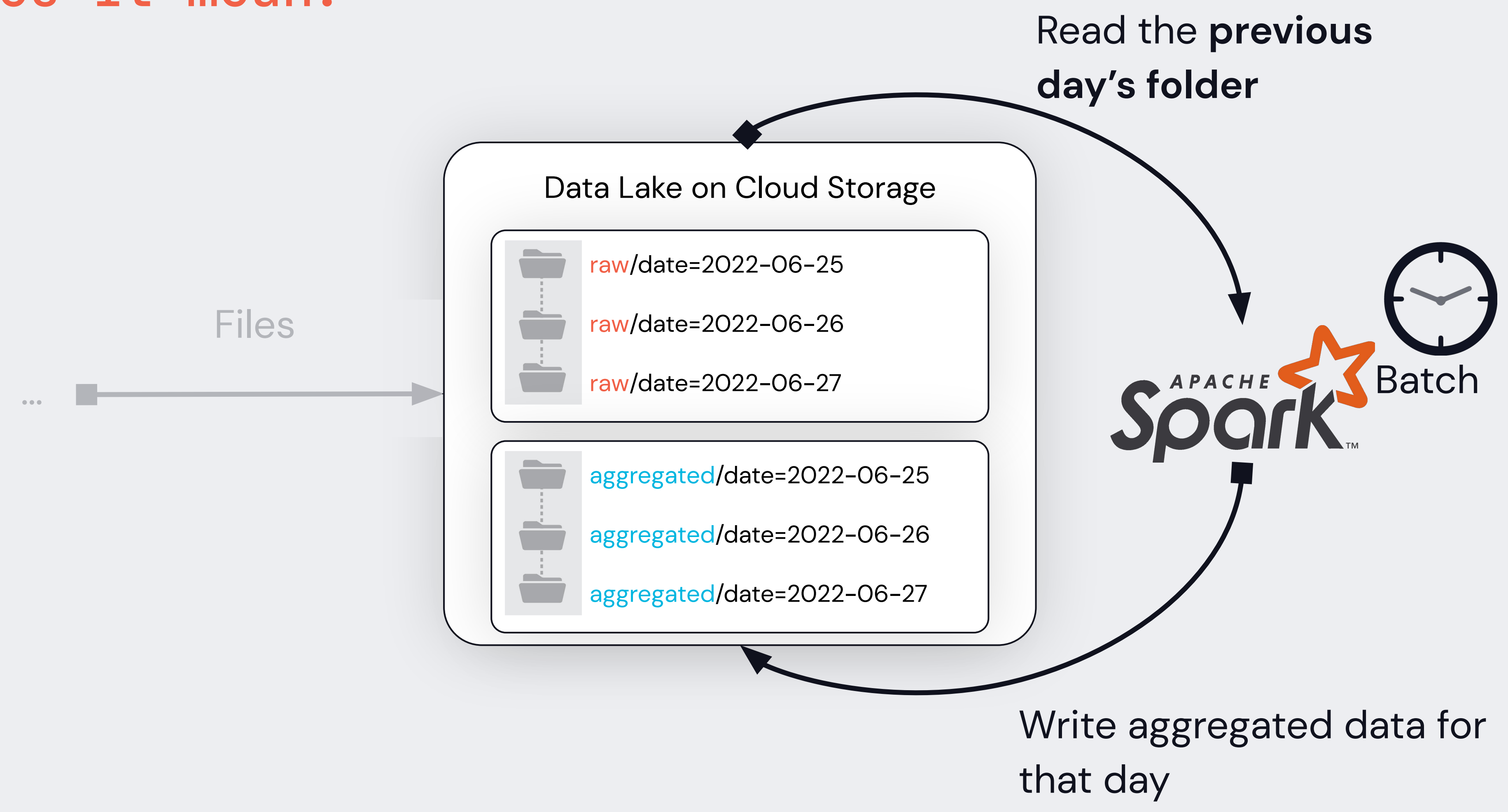
Batch Processing From a Data Lake

What does it mean?



Batch Processing From a Data Lake

What does it mean?



Batch Processing From a Data Lake

👍 It can **reduce operational costs**

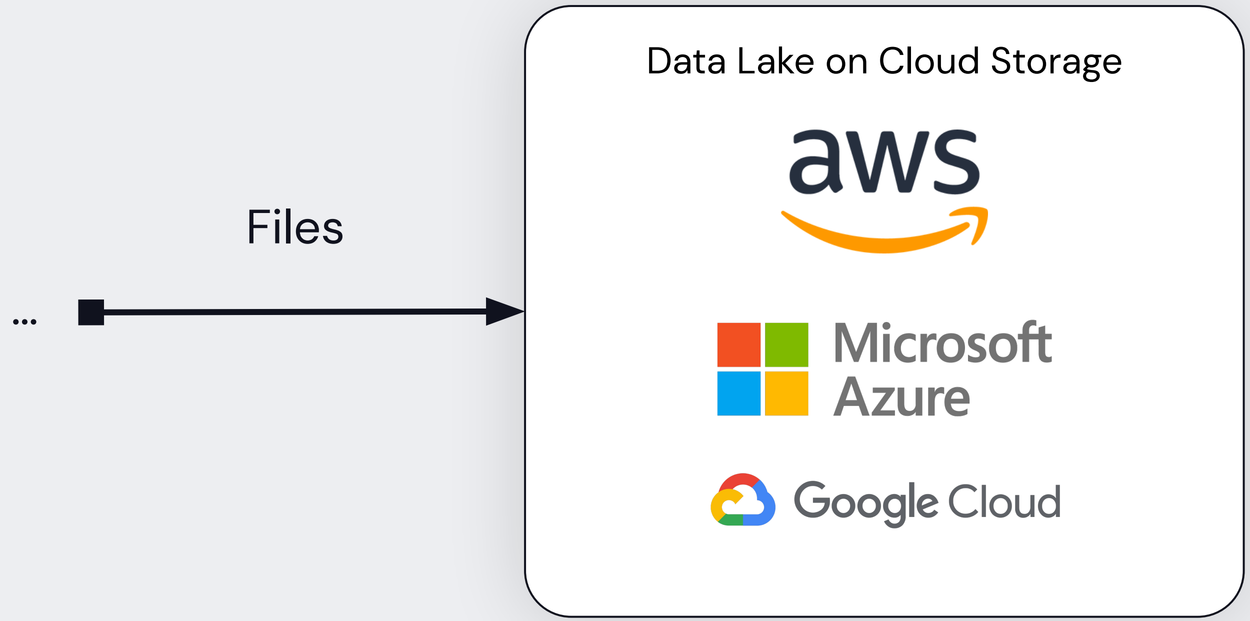
- By consuming **read-optimized** data (e.g Parquet files) from infinite-scale cloud storage
- By periodically launching **transient** Batch job clusters

Batch Processing From a Data Lake

- It can **reduce operational costs**
  - By consuming **read-optimized** data (e.g Parquet files) from infinite-scale cloud storage
  - By periodically launching **transient** Batch job clusters
- Hard to tell when all the raw data has arrived to the destination (=date) folder
  - How much time should we wait for the data to arrive? 2 hours? 6 hours? More?
    - If we don't wait long enough - we'll **miss late-arriving data**
    - If we re-process the same date folder to handle late-arriving data - we'll need to **add support for data mutability**

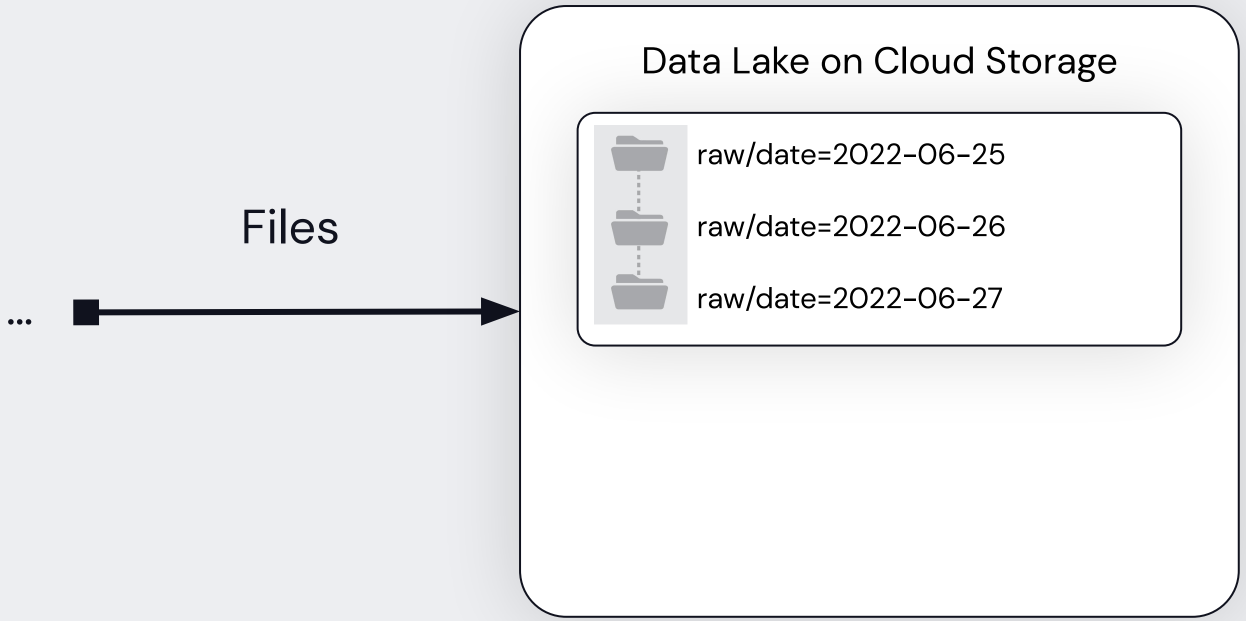
Stream Processing Over a Data Lake

What does it mean?



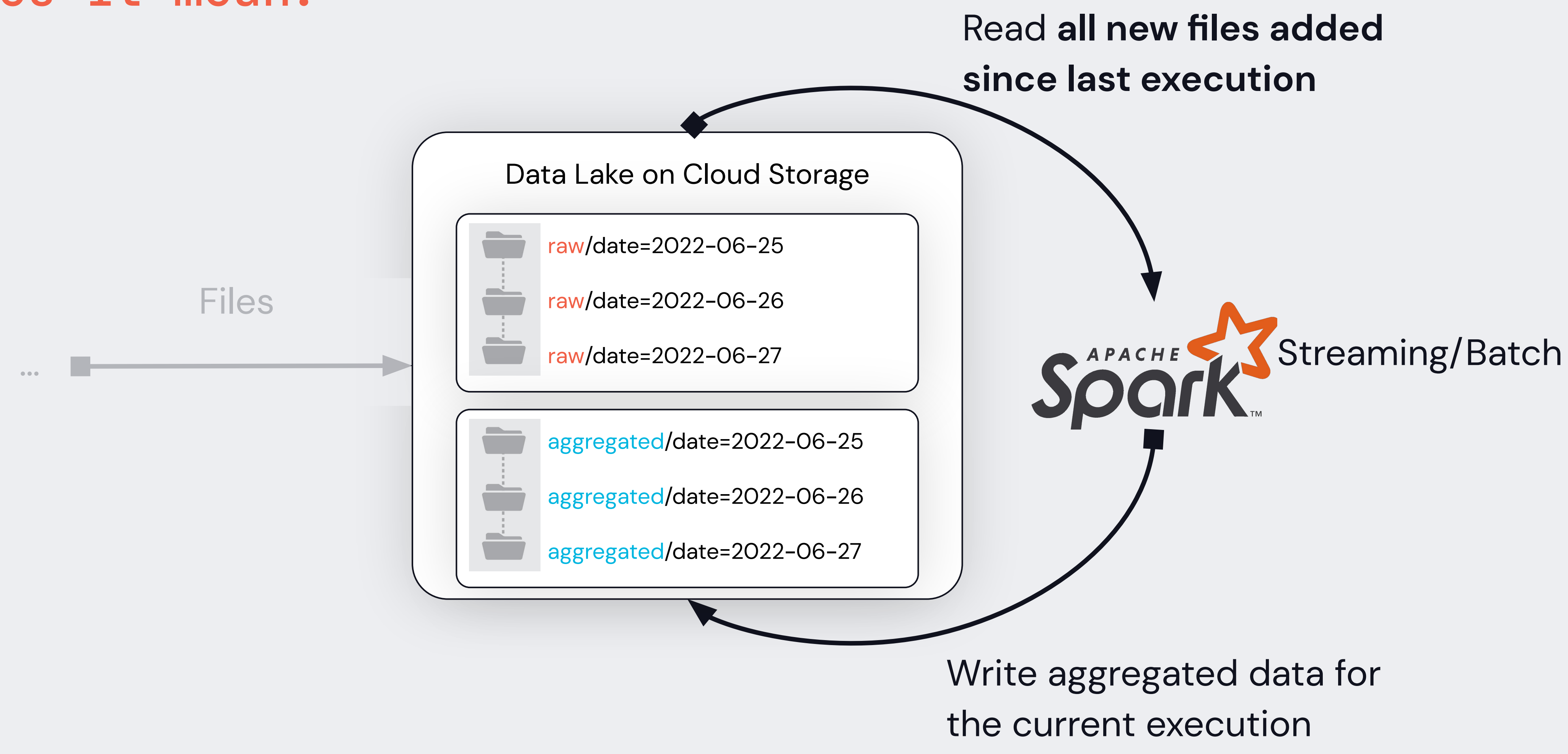
Stream Processing Over a Data Lake

What does it mean?



Stream Processing Over a Data Lake

What does it mean?



Stream Processing Over a Data Lake

👍 It can **reduce operational costs**

- By consuming **read-optimized** data (e.g Parquet files) from infinite-scale cloud storage
- By periodically launching **transient** Batch job clusters

Stream Processing Over a Data Lake

- It can **reduce operational costs**
  - By consuming **read-optimized** data (e.g Parquet files) from infinite-scale cloud storage
  - By periodically launching **transient** Batch job clusters
- Enables **handling late-arriving data**
  - By reading all new files since last execution (rather than reading a specific date folder)

Stream Processing Over a Data Lake

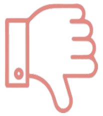
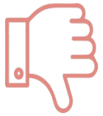
- 👍 It can **reduce operational costs**
  - By consuming **read-optimized** data (e.g Parquet files) from infinite-scale cloud storage
  - By periodically launching **transient** Batch job clusters
- 👍 Enables **handling late-arriving data**
  - By reading all new files since last execution (rather than reading a specific date folder)
- 👎 Lack of support for reading files in a streaming fashion

Stream Processing Over a Data Lake

- It can **reduce operational costs**
  - By consuming **read-optimized** data (e.g Parquet files) from infinite-scale cloud storage
  - By periodically launching **transient** Batch job clusters
- Enables **handling late-arriving data**
  - By reading all new files since last execution (rather than reading a specific date folder)
- Lack of support for reading files in a streaming fashion
- No built-in support for data mutability

3 Methods of Data Processing

A look back to 2019

	Stream processing into a Data Lake (for stateful apps)
Operational costs	
Burden on source systems (e.g Kafka brokers)	
Handling late-arriving data	
Support for data mutability	
Support for reading files in a streaming fashion	N/A

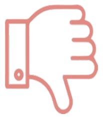
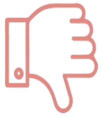
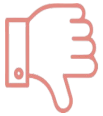
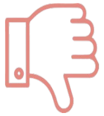
3 Methods of Data Processing

A look back to 2019

	Stream processing **into** a Data Lake (for stateful apps)	Batch processing **from** a Data Lake
Operational costs	👎	👍
Burden on source systems (e.g Kafka brokers)	👎	👍
Handling late-arriving data	👍	👎
Support for data mutability	👎	👎
Support for reading files in a streaming fashion	N/A	N/A

3 Methods of Data Processing

A look back to 2019

	Stream processing **into** a Data Lake (for stateful apps)	Batch processing **from** a Data Lake	Stream processing **over** a Data Lake
Operational costs			
Burden on source systems (e.g Kafka brokers)			
Handling late-arriving data			
Support for data mutability			
Support for reading files in a streaming fashion	N/A	N/A	

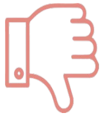
3 Methods of Data Processing

A look back to 2019

	Stream processing **into** a Data Lake (for stateful apps)	Batch processing **from** a Data Lake	Stream processing **over** a Data Lake
Operational costs	👎	👍	👍
Burden on source systems (e.g Kafka brokers)	👎	👍	👍
Handling late-arriving data	👍	👎	👍
Support for data mutability	👎	👎	👎
Support for reading files in a streaming fashion	N/A	N/A	👎

3 Methods of Data Processing

A look back to 2019

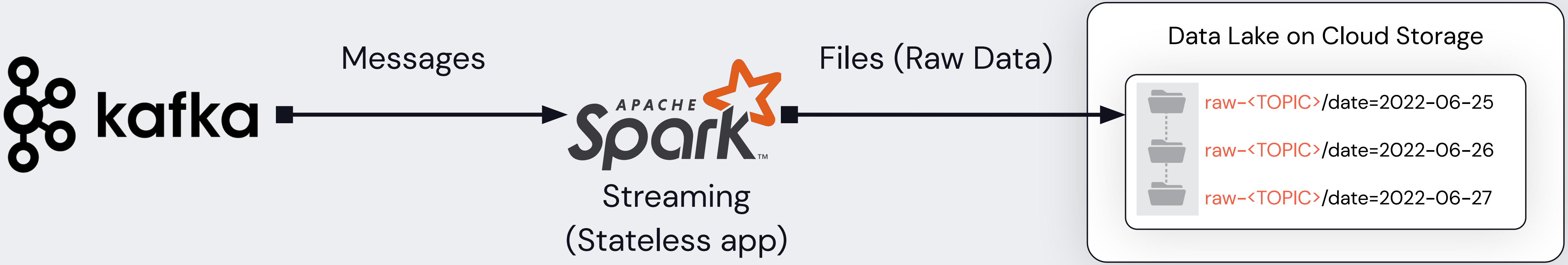
	Stream processing **into** a Data Lake (for stateful apps)	Batch processing **from** a Data Lake	Stream processing **over** a Data Lake
Operational costs			
Burden on source systems (e.g Kafka brokers)			
Handling late-arriving data			
Support for data mutability			
Support for reading files in a streaming fashion	N/A	N/A	

Stream Processing Over a Data Lake

DIY solution - Nielsen Marketing Cloud example

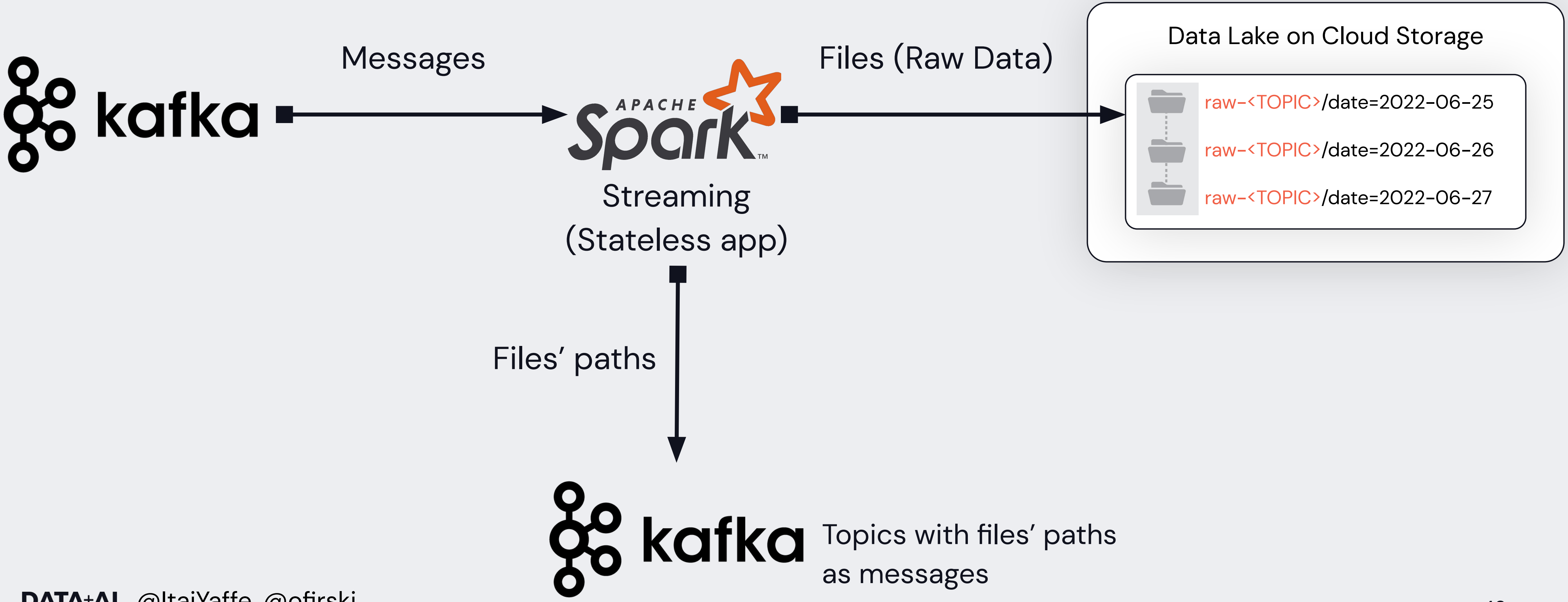
Stream Processing Over a Data Lake

DIY solution - producer side



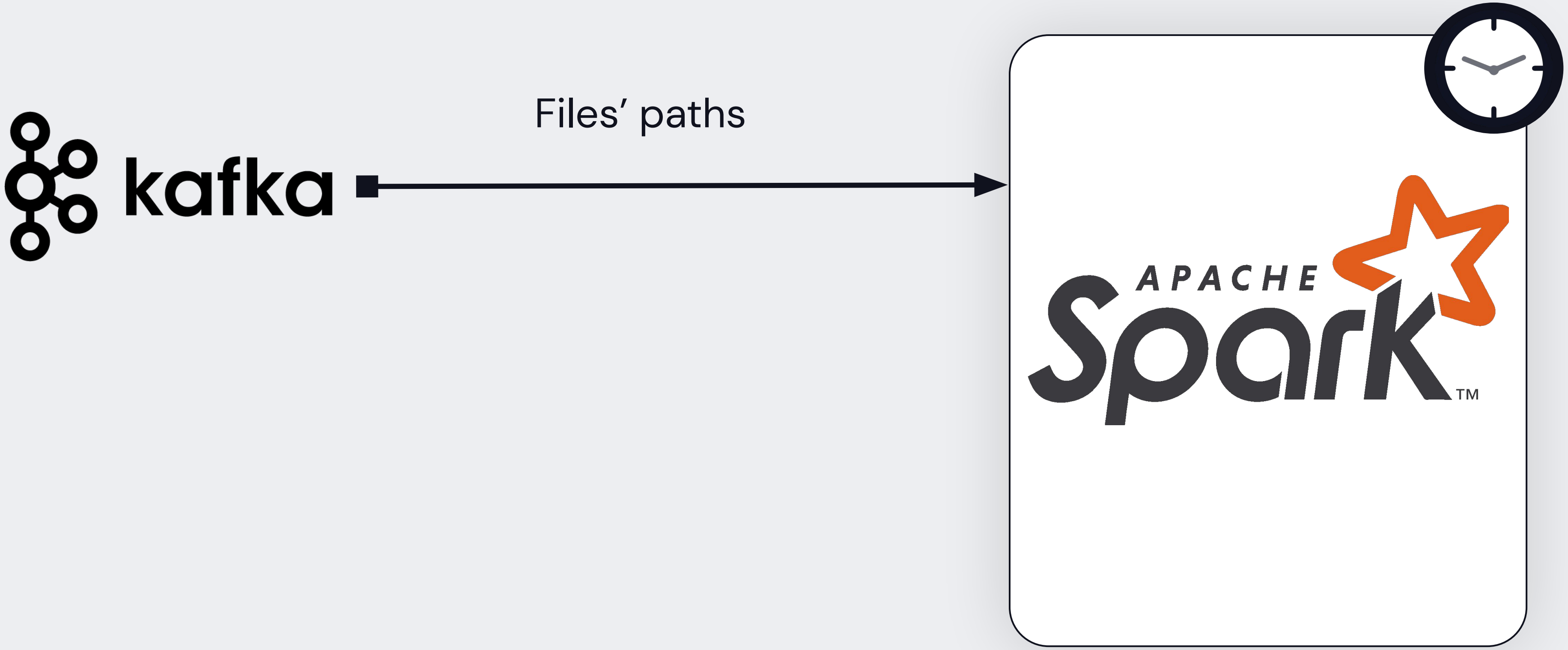
Stream Processing Over a Data Lake

DIY solution - producer side



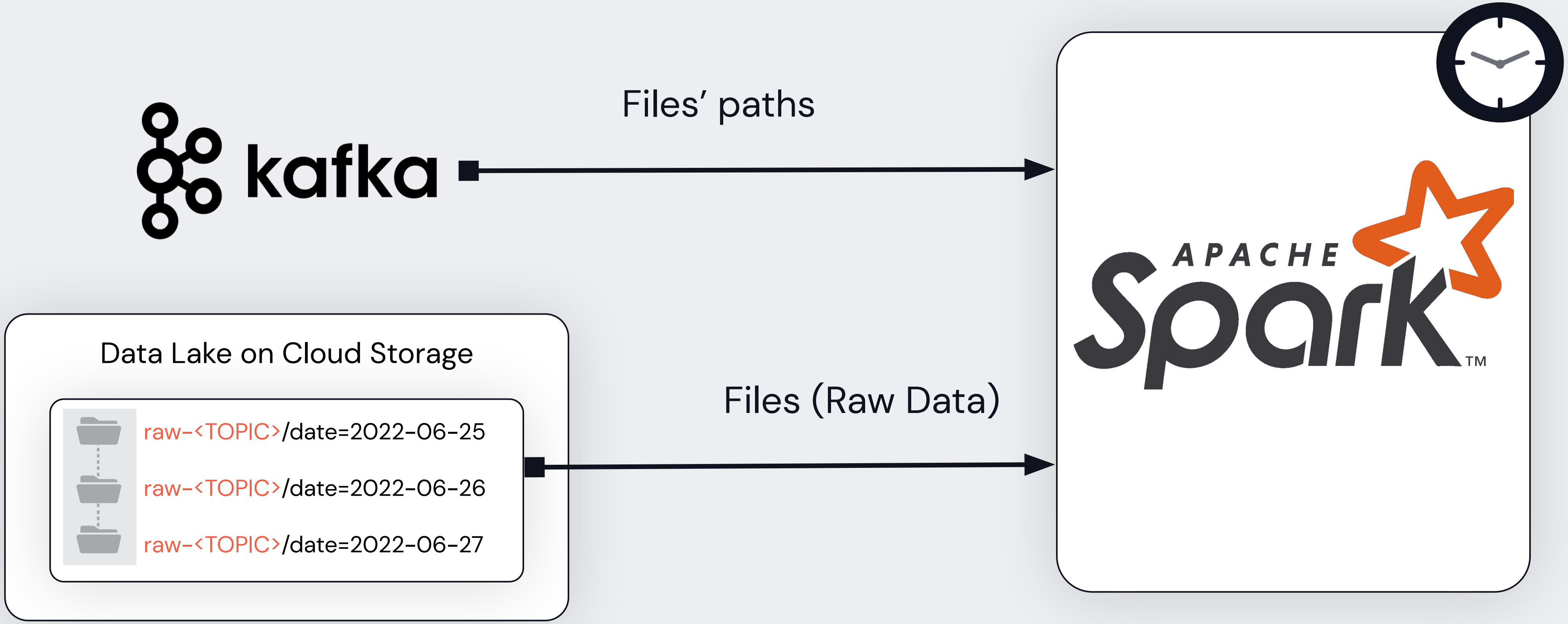
Stream Processing Over a Data Lake

DIY solution - consumer side



Stream Processing Over a Data Lake

DIY solution - consumer side

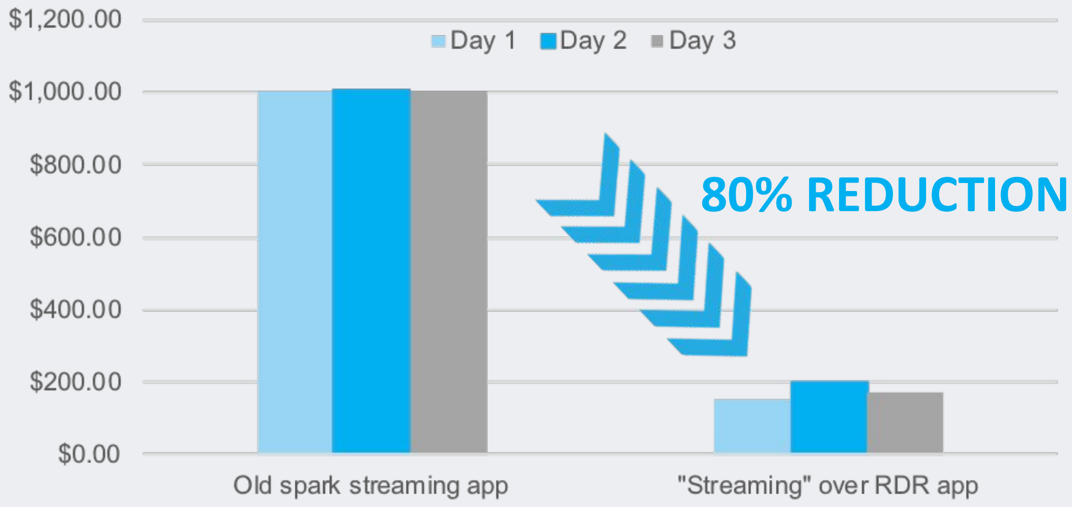


Stream Processing Over a Data Lake

DIY solution - Nielsen Marketing Cloud example

%80 **cost reduction**

- By Moving from **24/7** Spark Streaming clusters to **transient** Spark Batch clusters



Source: tinyurl.com/4s79mdpm (slide 33)

Stream Processing Over a Data Lake

DIY solution - Nielsen Marketing Cloud example

- %80 **cost reduction**
  - By Moving from **24/7** Spark Streaming clusters to **transient** Spark Batch clusters
- No more extreme load on Kafka brokers
  - Only **one consumer per Kafka topic** (a stateless Spark Streaming application)
  - **All other consumers read from S3** (i.e our Data Lake)

Stream Processing Over a Data Lake

DIY solution - Nielsen Marketing Cloud example

- %80 **cost reduction**
  - By Moving from **24/7** Spark Streaming clusters to **transient** Spark Batch clusters
- No more extreme load on Kafka brokers
  - Only **one consumer per Kafka topic** (a stateless Spark Streaming application)
  - **All other consumers read from S3** (i.e our Data Lake)
- Built-in **handling of late-arriving** events
  - Instead of reading entire folders by date (e.g date=2022-06-27), we **read by file names**, which means **we don’t miss files** that were written to a folder after that date has passed

Stream Processing Over a Data Lake

DIY solution - Nielsen Marketing Cloud example

- But... It took **a small team and a few months of development** to build this **proprietary** solution

Fast Forward to 2022...

What has changed?

- With the rise of tools like [Delta Lake] and features such as [Auto Loader], this becomes SOOOO much easier

Auto Loader

By Databricks

- Incrementally and efficiently processes new data files as they arrive in cloud storage like AWS S3
- Supports multiple file formats
- Provides a Structured Streaming source called `cloudFiles` which automatically processes new files as they arrive
- Enables the developer to incrementally write the data to any supported sink

Auto Loader

Modes for detecting new files

- Directory listing
  - Auto Loader identifies new files by listing the input directory in an optimized manner

Auto Loader

Modes for detecting new files

- Directory listing
  - Auto Loader identifies new files by listing the input directory in an optimized manner
- File notification
  - Auto Loader can automatically set up a notification service (e.g AWS SNS) and queue service (e.g AWS SQS) that subscribe to file events from the input directory
  - This mode is more performant and scalable for large input directories or a high volume of files

Delta Lake

An open-source storage framework

- Brings reliability to data lakes
  - ACID transactions
  - Support for data mutability
  - Ability to “time travel”
  - Scalable metadata handling
  - Unifies streaming and batch data processing

Delta Lake

An open-source storage framework

- An open-source format
  - Uses versioned Parquet files to store the data
  - Also stores a transaction log to keep track of all the commits made to the table or blob store directory to provide ACID transactions

Delta Lake

An open-source storage framework

- Has a large ecosystem
  - Can be integrated with Compute engines including Spark, PrestoDB, Flink, Trino, and Hive
  - Provides APIs for Scala, Java, Rust, Ruby, and Python

Fast Forward to 2022...

What has changed?

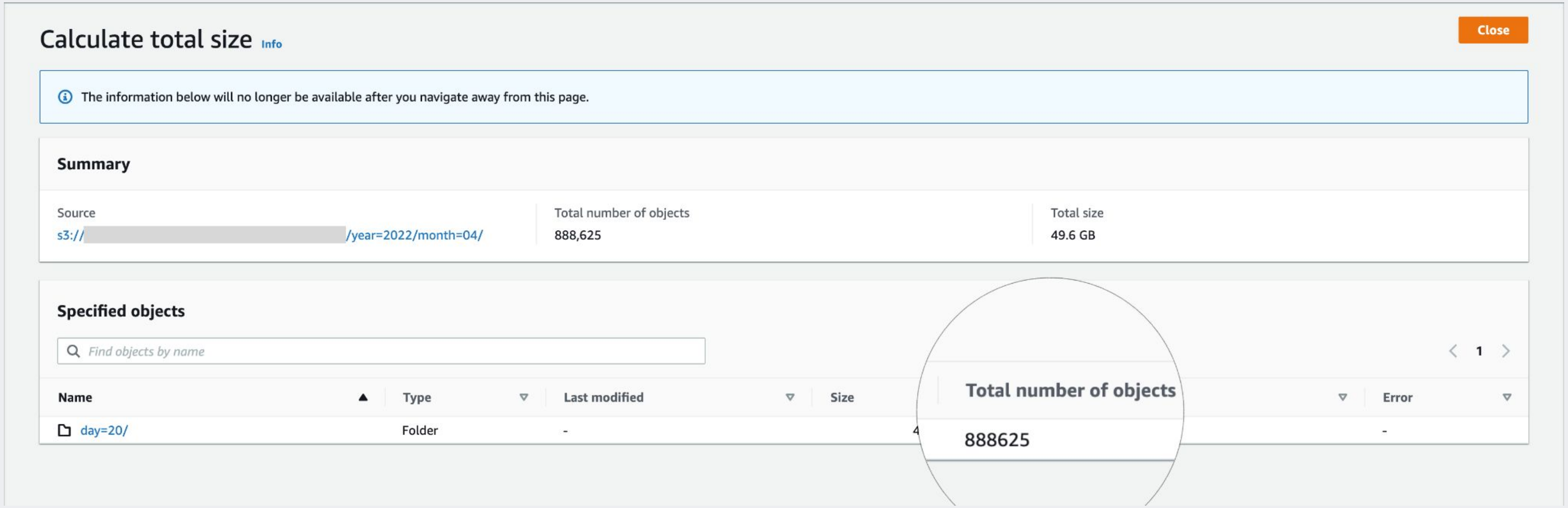
- With the rise of tools like Delta Lake and features such as Auto Loader, this becomes SOOOO much easier
  - 👍 **Delta Lake** brings **support for data mutability** on top of your Data Lake
  - 👍 **Auto Loader** allows you to **read files in a streaming fashion** in a few lines of code
    - Behind the scenes, it actually uses a similar mechanism to what we implemented at Nielsen

Now Let’s “Time-Travel” Forward to 2022

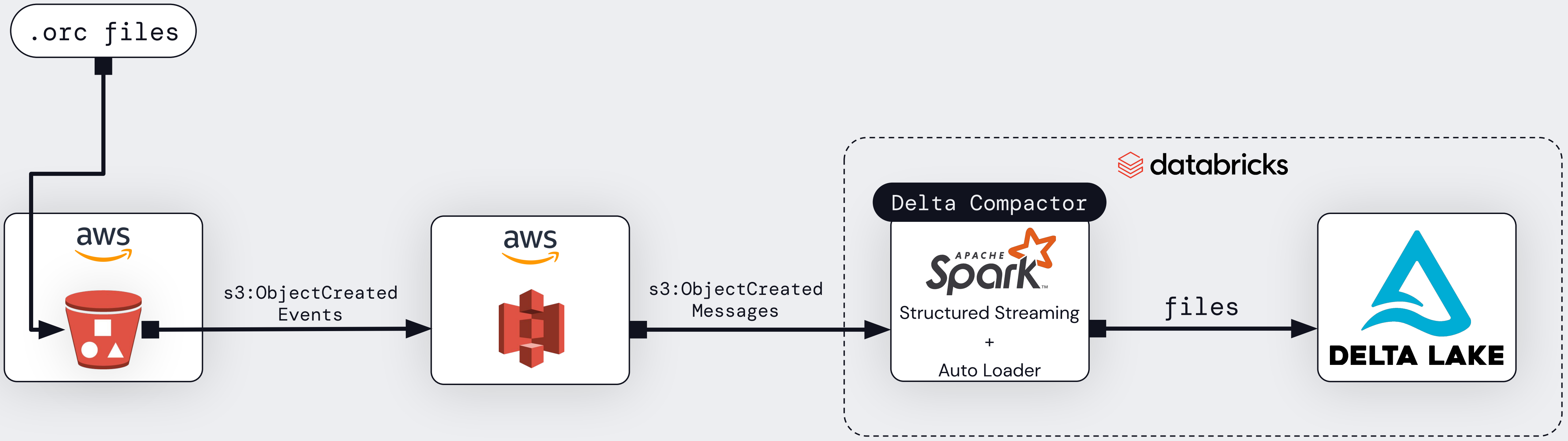
The Small Files Challenge

Recap

- In order to keep our Data Lake optimized, we need to compact **a stream of small files**



Introducing Nexar’s Delta Compactor



Delta Compactor Code Highlights

Setup Source

```
def setup_stream_source(self) -> DataFrame:
    return self.ctx.spark.readStream.format("cloudFiles") \
        .option("cloudFiles.useNotifications", "true") \
        .option("cloudFiles.includeExistingFiles", "false") \
        .option("cloudFiles.format", self.job_config.source_files_format) \
        .option("cloudFiles.maxFilesPerTrigger", self.job_config.max_files_per_trigger) \
        .option("cloudFiles.maxBytesPerTrigger", self.job_config.max_bytes_per_trigger) \
        .option("cloudFiles.queueUrl", self.job_config.queue_url) \
        .schema(self.get_schema()) \
        .load(self.job_config.source_files_path)
```

Delta Compactor Code Highlights

Setup Source

Structured Streaming Source

```
def setup_stream_source(self) -> DataFrame:
    return self.ctx.spark.readStream.format("cloudFiles") \
        .option("cloudFiles.useNotifications", "true") \
        .option("cloudFiles.includeExistingFiles", "false") \
        .option("cloudFiles.format", self.job_config.source_files_format) \
        .option("cloudFiles.maxFilesPerTrigger", self.job_config.max_files_per_trigger) \
        .option("cloudFiles.maxBytesPerTrigger", self.job_config.max_bytes_per_trigger) \
        .option("cloudFiles.queueUrl", self.job_config.queue_url) \
        .schema(self.get_schema()) \
        .load(self.job_config.source_files_path)
```

Delta Compactor Code Highlights

Setup Source

Uses `s3:ObjectCreated` Events

```python
def setup_stream_source(self) -> DataFrame:
    return self.ctx.spark.readStream.format("cloudFiles") \
        .option("cloudFiles.useNotifications", "true") \
        .option("cloudFiles.includeExistingFiles", "false") \
        .option("cloudFiles.format", self.job_config.source_files_format) \
        .option("cloudFiles.maxFilesPerTrigger", self.job_config.max_files_per_trigger) \
        .option("cloudFiles.maxBytesPerTrigger", self.job_config.max_bytes_per_trigger) \
        .option("cloudFiles.queueUrl", self.job_config.queue_url) \
        .schema(self.get_schema()) \
        .load(self.job_config.source_files_path)
```

Delta Compactor Code Highlights

Setup Source

Don’t include existing files

```python
def setup_stream_source(self) -> DataFrame:
   return self.ctx.spark.readStream.format("cloudFiles") \
       .option("cloudFiles.useNotifications", "true") \
       .option("cloudFiles.includeExistingFiles", "false") \
       .option("cloudFiles.format", self.job_config.source_files_format) \
       .option("cloudFiles.maxFilesPerTrigger", self.job_config.max_files_per_trigger) \
       .option("cloudFiles.maxBytesPerTrigger", self.job_config.max_bytes_per_trigger) \
       .option("cloudFiles.queueUrl", self.job_config.queue_url) \
       .schema(self.get_schema()) \
       .load(self.job_config.source_files_path)
```

Delta Compactor Code Highlights

Setup Source

Set source files format (ORC)

```
def setup_stream_source(self) -> DataFrame:
    return self.ctx.spark.readStream.format("cloudFiles") \
        .option("cloudFiles.useNotifications", "true") \
        .option("cloudFiles.includeExistingFiles", "false") \
        .option("cloudFiles.format", self.job_config.source_files_format) \
        .option("cloudFiles.maxFilesPerTrigger", self.job_config.max_files_per_trigger) \
        .option("cloudFiles.maxBytesPerTrigger", self.job_config.max_bytes_per_trigger) \
        .option("cloudFiles.queueUrl", self.job_config.queue_url) \
        .schema(self.get_schema()) \
        .load(self.job_config.source_files_path)
```

Delta Compactor Code Highlights

Setup Source

Set limits per trigger

```
def setup_stream_source(self) -> DataFrame:
    return self.ctx.spark.readStream.format("cloudFiles") \
        .option("cloudFiles.useNotifications", "true") \
        .option("cloudFiles.includeExistingFiles", "false") \
        .option("cloudFiles.format", self.job_config.source_files_format) \
        .option("cloudFiles.maxFilesPerTrigger", self.job_config.max_files_per_trigger) \
        .option("cloudFiles.maxBytesPerTrigger", self.job_config.max_bytes_per_trigger) \
        .option("cloudFiles.queueUrl", self.job_config.queue_url) \
        .schema(self.get_schema()) \
        .load(self.job_config.source_files_path)
```

Delta Compactor Code Highlights

Setup Source

Set SQS queue url

```python
def setup_stream_source(self) -> DataFrame:
    return self.ctx.spark.readStream.format("cloudFiles") \
        .option("cloudFiles.useNotifications", "true") \
        .option("cloudFiles.includeExistingFiles", "false") \
        .option("cloudFiles.format", self.job_config.source_files_format) \
        .option("cloudFiles.maxFilesPerTrigger", self.job_config.max_files_per_trigger) \
        .option("cloudFiles.maxBytesPerTrigger", self.job_config.max_bytes_per_trigger) \
        .option("cloudFiles.queueUrl", self.job_config.queue_url) \
        .schema(self.get_schema()) \
        .load(self.job_config.source_files_path)
```

Delta Compactor Code Highlights

Setup Source

Specify source files schema

```
def setup_stream_source(self) -> DataFrame:
    return self.ctx.spark.readStream.format("cloudFiles") \
        .option("cloudFiles.useNotifications", "true") \
        .option("cloudFiles.includeExistingFiles", "false") \
        .option("cloudFiles.format", self.job_config.source_files_format) \
        .option("cloudFiles.maxFilesPerTrigger", self.job_config.max_files_per_trigger) \
        .option("cloudFiles.maxBytesPerTrigger", self.job_config.max_bytes_per_trigger) \
        .option("cloudFiles.queueUrl", self.job_config.queue_url) \
        .schema(self.get_schema()) \
        .load(self.job_config.source_files_path)
```

Delta Compactor Code Highlights

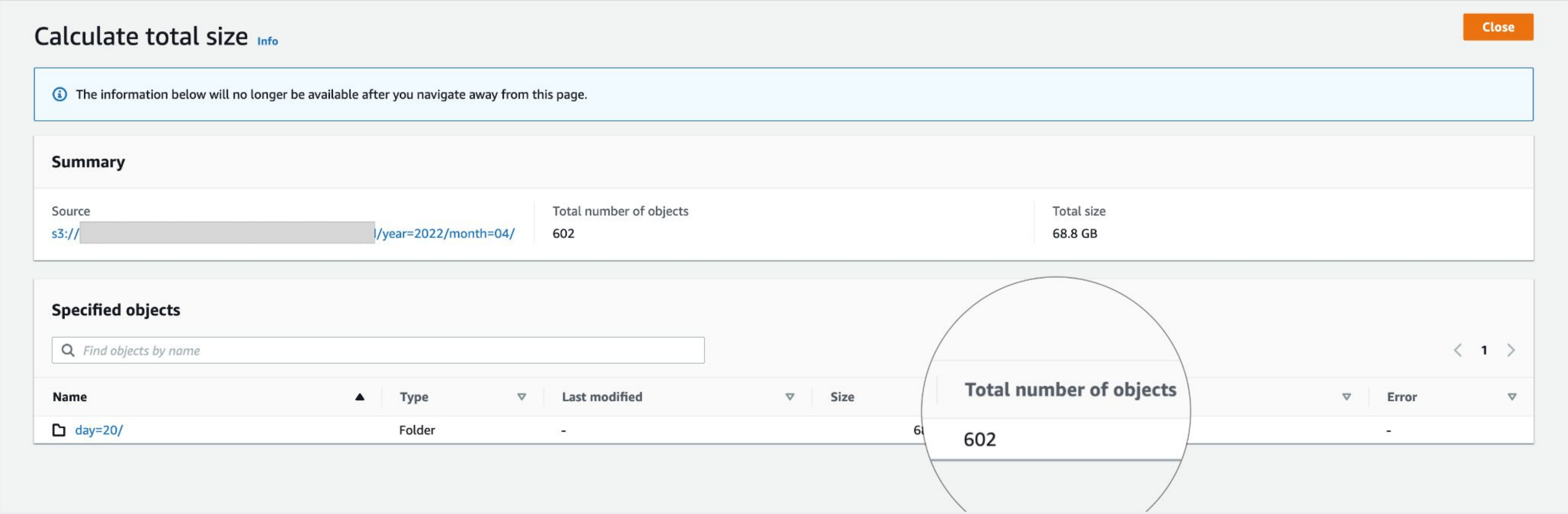
Setup Sink

```python
def setup_stream_sink(self, stream: DataFrame):
    write_stream = stream.writeStream.format("delta") \
        .outputMode("append") \
        .option("checkpointLocation", self.job_config.delta_table_checkpoint_path)
    return write_stream.table(self.job_config.table_name)
```

Delta Compactor

Summary

Combining Delta Lake's auto compaction with Databricks' Auto Loader, creates a reliable, simple and cost-effective solution.

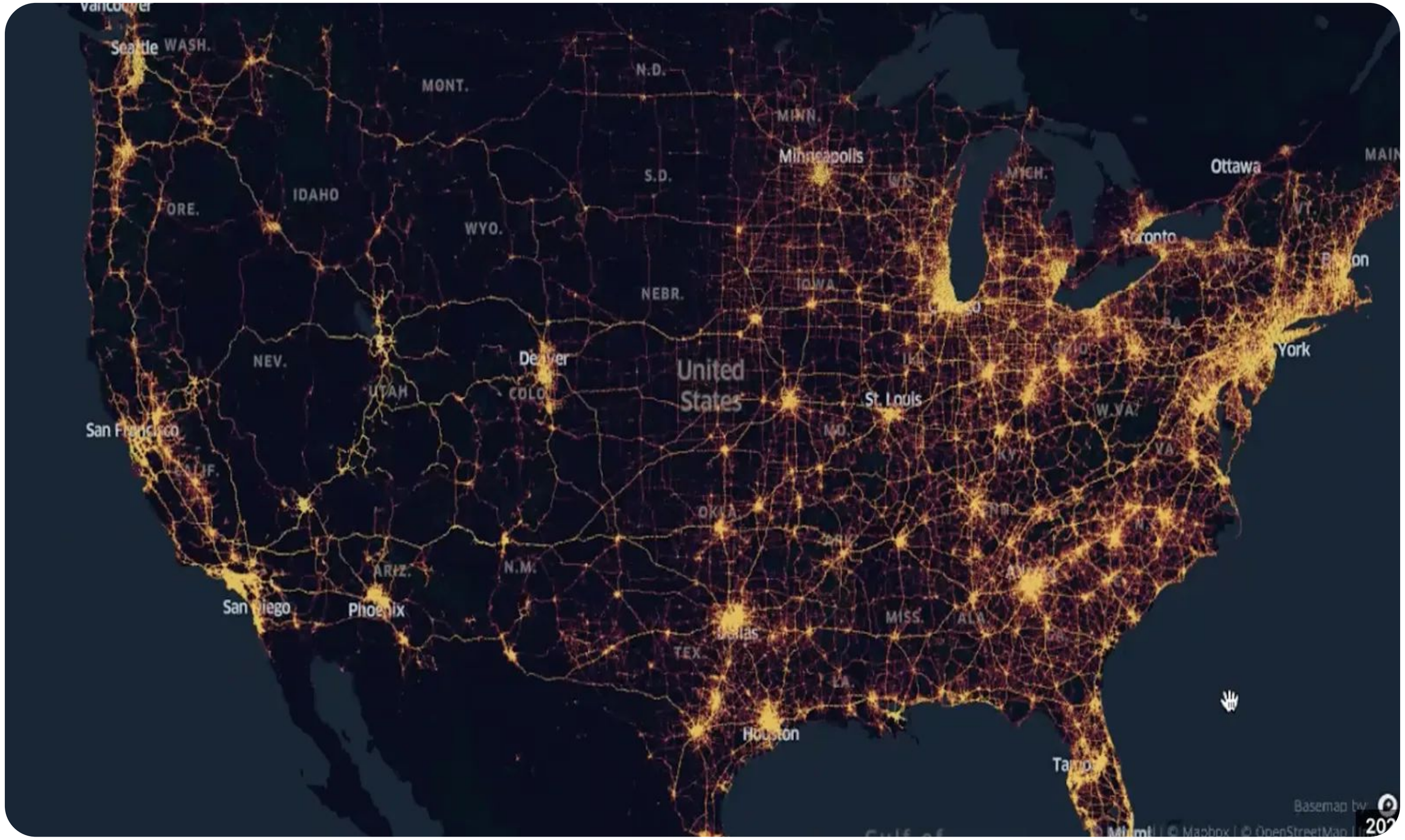



MUTABLE DATA
ME
APPEND-ONLY DATA
imgflip.com

Constant Coverage Index

Data Mutability at Scale

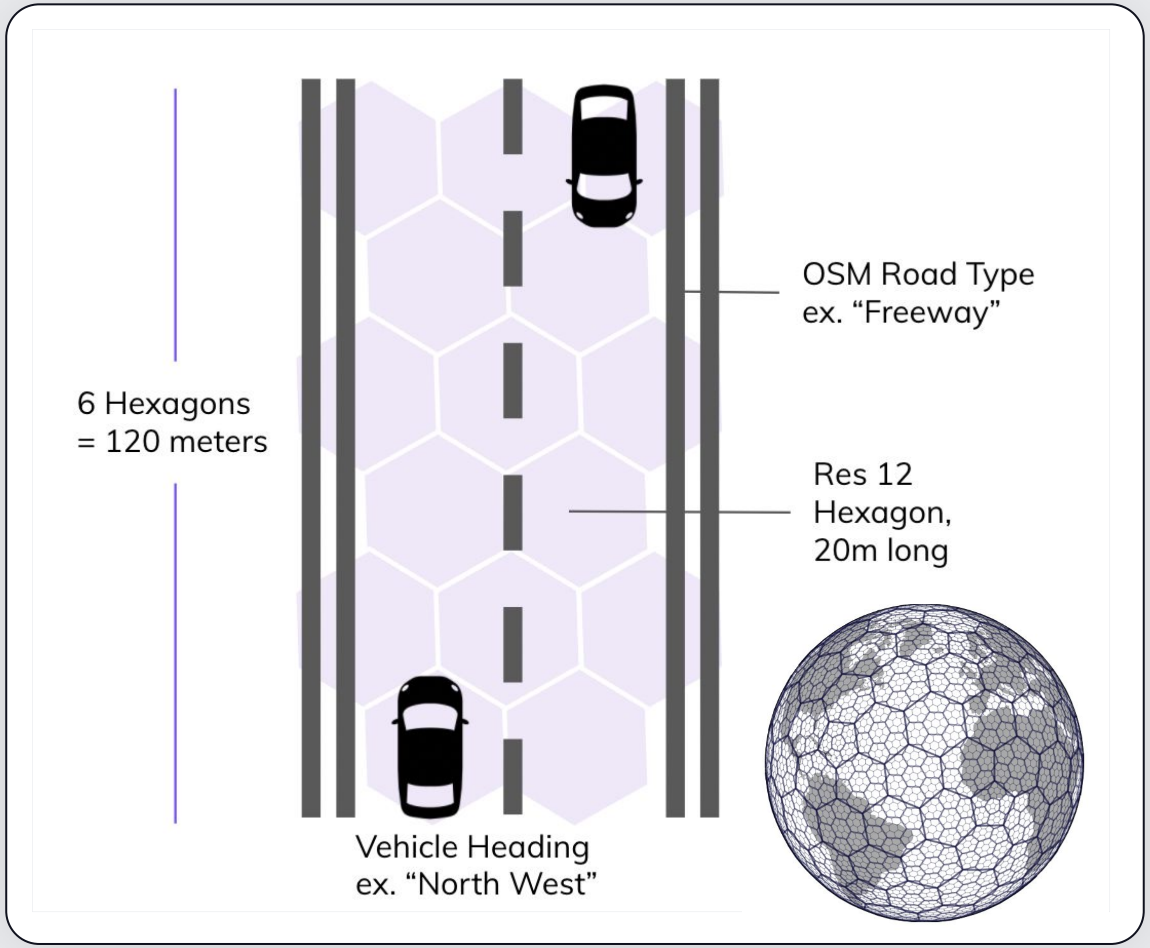
Providing a constant, predictable and consistent view of the roads



Constant Coverage

Data Mutability at Scale

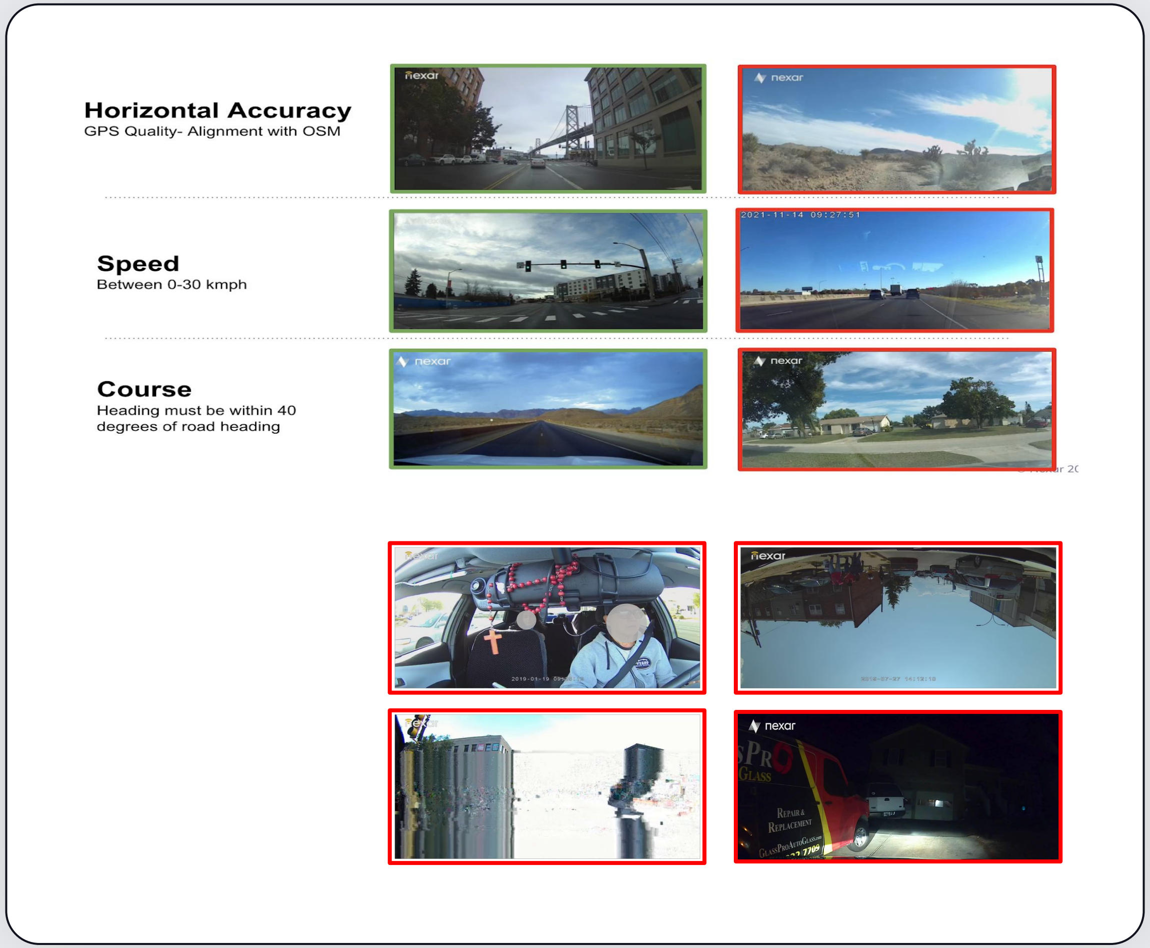
- Always keep the "best" N images for every spatial area
- The spatial area is called HexSeg: a combination of "Hexagon" and "Road Segment"



Constant Coverage

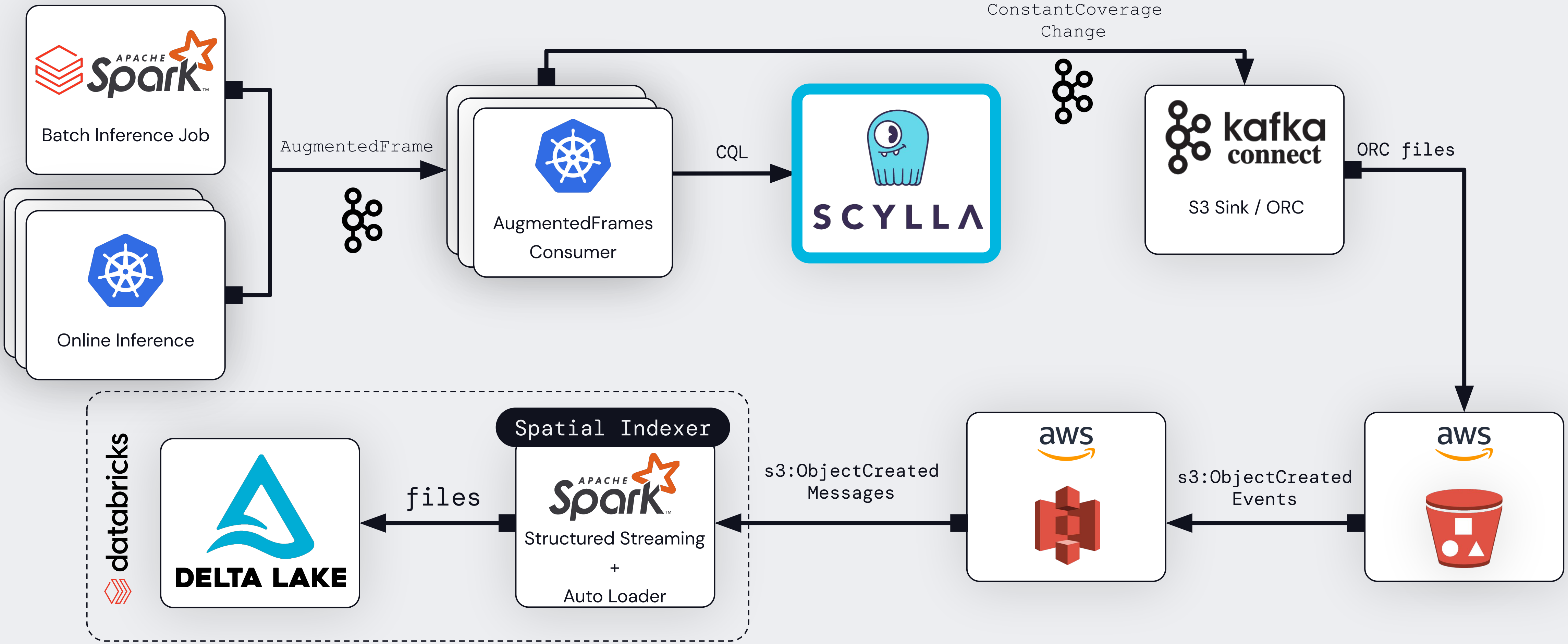
Data Mutability at Scale

- For each HexSeg we're maintaining a stack of frames, limited by a pre-defined cap
- Once the stack is filled up it will rotate and frames considered as "better" will replace existing frames within the HexSeg



Constant Coverage – Initial Version

Scylla as Source of Truth



Constant Coverage – Initial Version

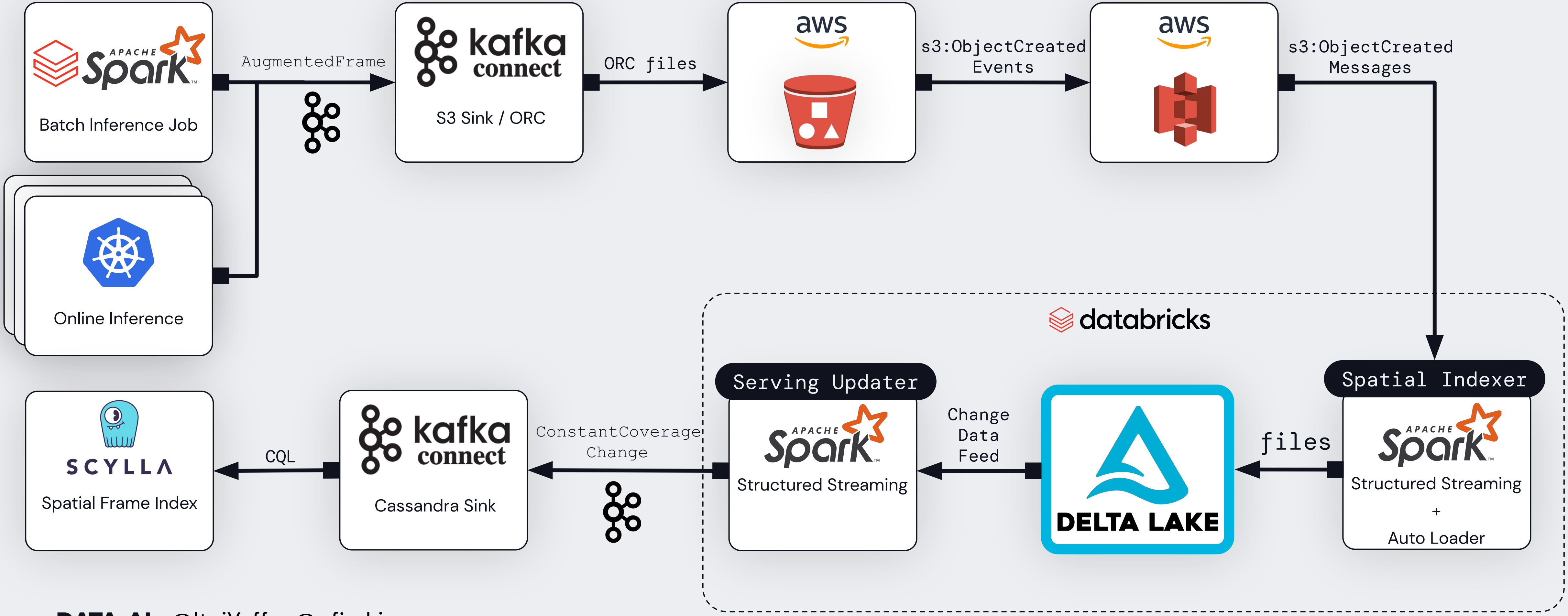
Scylla as Source of Truth

- Low latency updates of the Constant Coverage index
- Straightforward and simple logic

- Serving store cluster's cost is linearly dependant on storage size
- Write throughput is bounded to the serving store's capacity
- Difficult to debug

Constant Coverage – Final Version

Delta as Source of Truth



Constant Coverage – Final Version

Delta as Source of Truth

- Serving store cluster's cost is only dependant on clients demand
- Write throughput is bounded to the serving store updater job
- Still straightforward but adds some complexity with Change Data Feed
- Easy to debug issues

Summary

We all have a few burns from keeping our Data Lakes optimized…

Summary

We all have a few burns from keeping our Data Lakes optimized…

So don't reinvent the wheel –

Summary

We all have a few burns from keeping our Data Lakes optimized…

So don't reinvent the wheel –

Leverage existing tools and practices

Summary

We all have a few burns from keeping our Data Lakes optimized...

So don't reinvent the wheel –

Leverage existing tools and practices, e.g:

- Delta Lake
  - Read-optimized format
  - Data mutability support

Summary

We all have a few burns from keeping our Data Lakes optimized...

So don't reinvent the wheel –

Leverage existing tools and practices, e.g:

- Delta Lake
  - Read-optimized format
  - Data mutability support
- Stream processing **over** your Data Lake
  - Can reduce operational costs
  - Potentially use Auto Loader

Want to know more?

- Women in Big Data
  - A world-wide program that aims:
    - To inspire, connect, grow and champion success of women in the Big Data & analytics field
  - 40+ chapters and 17,000+ members world-wide
  - Everyone can join (regardless of gender), so find a chapter near you – www.womeninbigdata.org/wibd-structure/

- Past and upcoming talks
  - Itai's Spark & Kafka talk (Spark+AI Summit 2019 Europe) – tinyurl.com/4s79mdpm
  - Delta Lake 2.0 by Tathagata Das & Denny Lee (Tuesday, 2:50PM) – tinyurl.com/57e8nf5f

- Resources
  - Nexar's Constant Coverage blog post – tinyurl.com/y98j4hw9
  - Delta Lake – delta.io
  - Databricks Auto Loader – tinyurl.com/3yj2srvx

DATA+AI SUMMIT 2022

Thank you

Itai Yaffe

Senior Solutions Architect, Databricks

Itai Yaffe @ItaiYaffe

Ofir Kerker

Data Platform Tech Lead, Nexar

Ofir Kerker @ofirski_