databricks

Data Engineering

with the Data Intelligence Platform

11 de setembro de 2024

Ricardo Conegliam - Sr. Specialist Solution Architect

Agenda

Data Engineering with the Data Intelligence Platform

Introduction

- Introduction to Lakehouse
- Apache Spark
- Delta Fundamentos
- Batch processing vs. stream processing
- Understanding Streaming Data
- Data Lake Challenges for Streaming Data
- Spark: Anatomy of a Streaming Query
- **Extra:** A quick look at Delta Live Tables

Data Operations on Streaming Data

- Ingesting Data with Auto Loader
- Working with Windows-based aggregations
- Handling late data with watermarks
- Streaming Multi-Hop Tables
- Stream-Static Joins

Updating Tables with Streaming Data

- Streaming Upserts
- Delta Change Data Feed

Advanced Topics

- Streaming Model Deployment with MLflow
- Streaming Analytics with Databricks SQL
- Production Best Practices

Bring your use case

- Use case discovery and open discussion

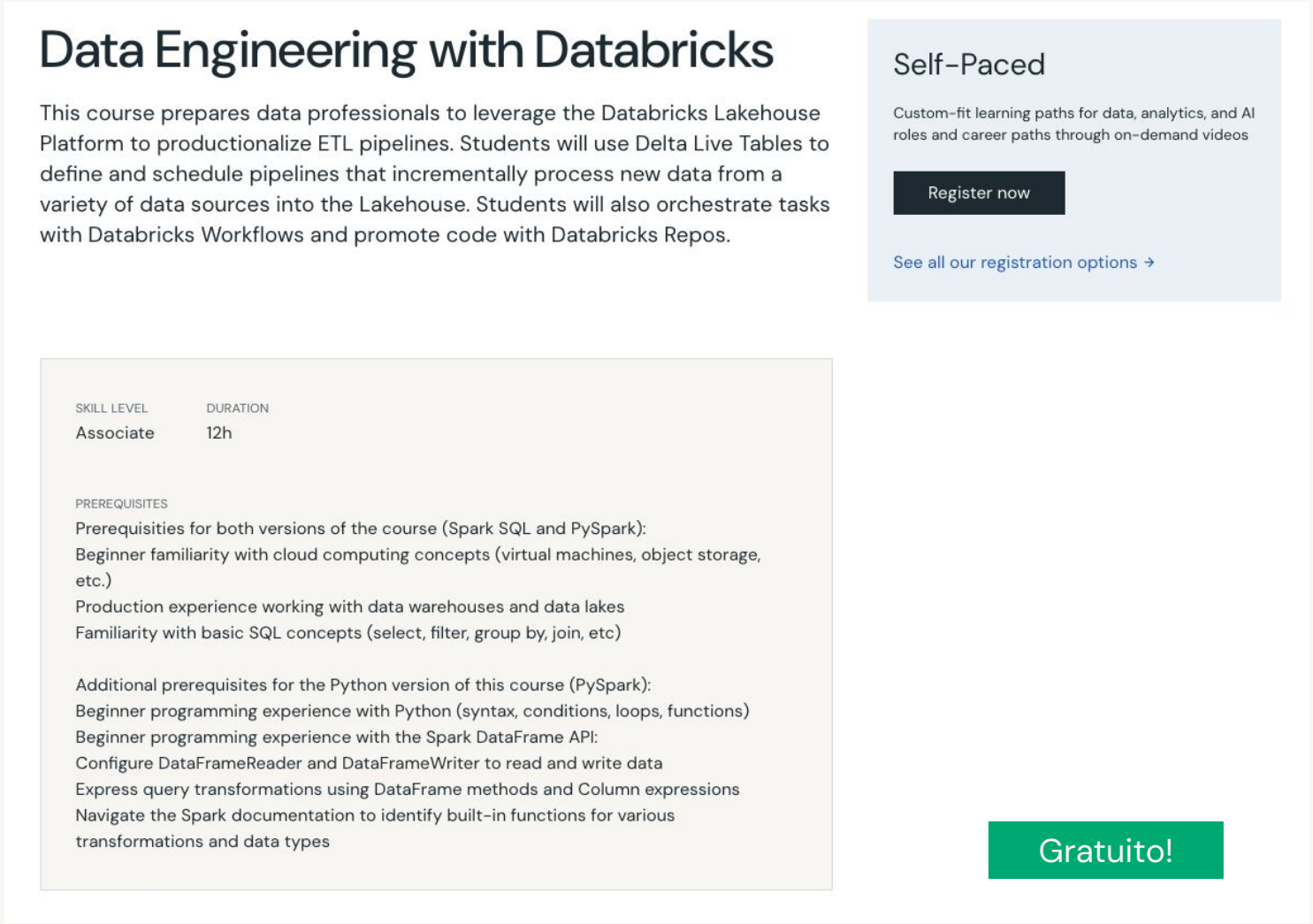
Objetivo

- Passar por um overview nos conceitos do Spark e Delta
- Apresentar as novidades do Delta
- Diferenciar processamento batch de streaming
- Apresentar boas práticas no processamento streaming
- Exercícios práticos para entender como trabalhar com engenharia de dados no Databricks, e como ela facilita o dia a dia dos engenheiros
- Levantamento de discussões para caso vocês vejam algumas dessas tecnologias serem aplicadas em projetos

Objetivo

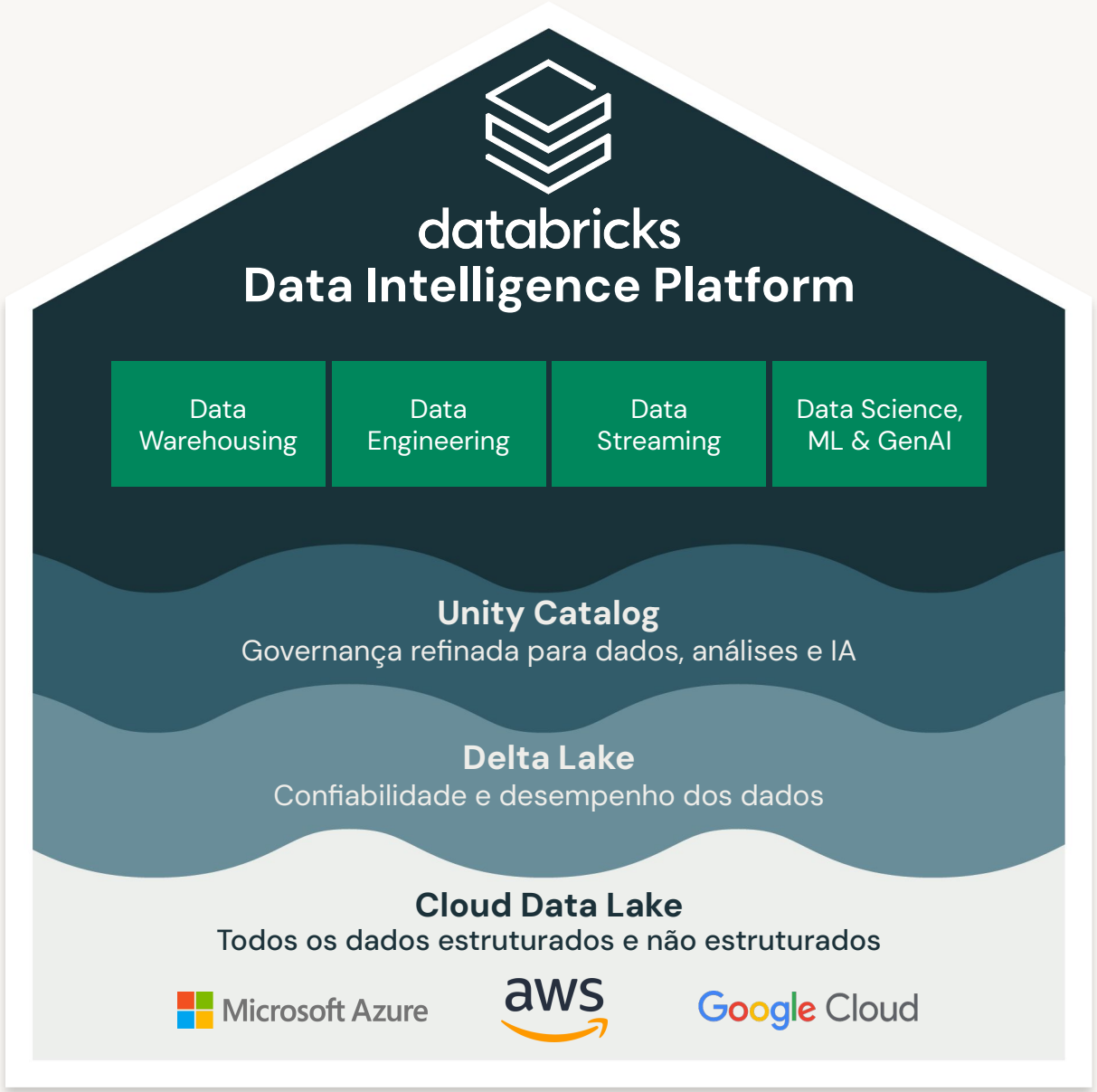
Não é objetivo desse workshop, tornar vocês expert em programação Spark, para isso, recomendamos o seguinte curso:

[Data Engineering with Databricks]



Introdução

O que é o Lakehouse?



Databricks Lakehouse Platform

Simples

Unifique seu data warehouse e IA casos de uso em uma única plataforma

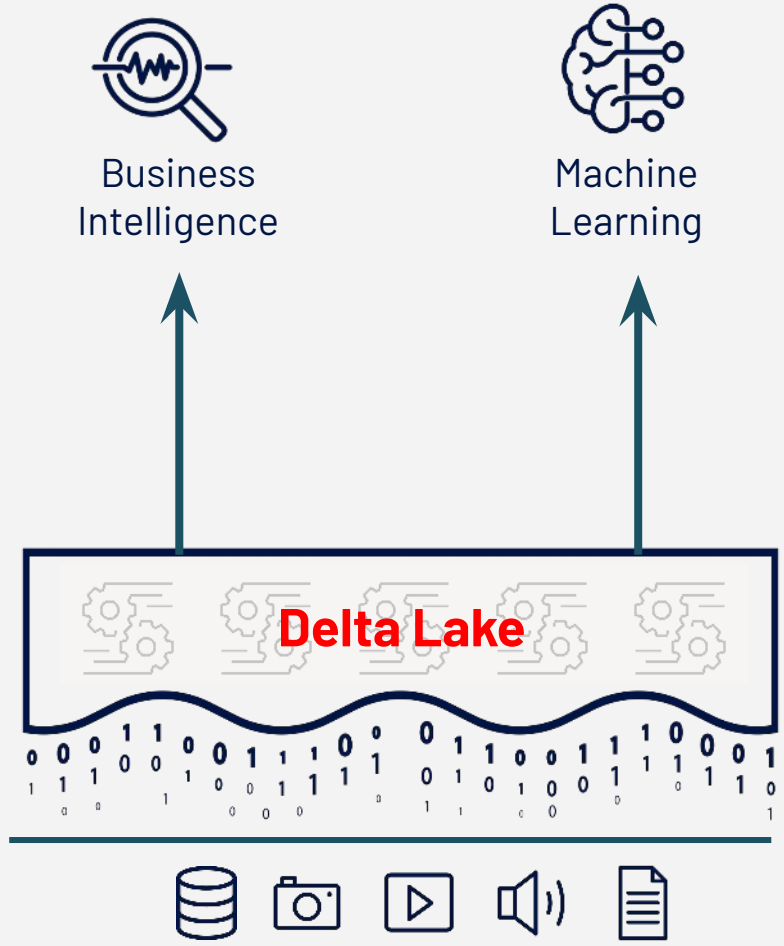
Multicloud

Uma plataforma de dados consistente em todas as nuvens

Open

Construído em código aberto e padrões abertos

Engenharia de Dados na Arquitetura Lakehouse



DELTA LAKE

Uma fundação dos os dados para BI, Data Science & ML

- Adiciona confiabilidade, desempenho, governança e qualidade aos data lakes existentes
- Baseado no formato de dados abertos (Parquet)
- Simplifica a engenharia de dados com uma abordagem de data lake com curadoria

Visão geral do Spark

Visão geral do Spark

Arquitetura Spark

O motor unificado padrão (de-Facto) para analytics e processamento de big data

O maior projeto Open-Source em tecnologia de processamento de dados

Criado pelos fundadores da Databricks durante seu doutorado em Berkeley

Spark Overview

Benefícios do Spark

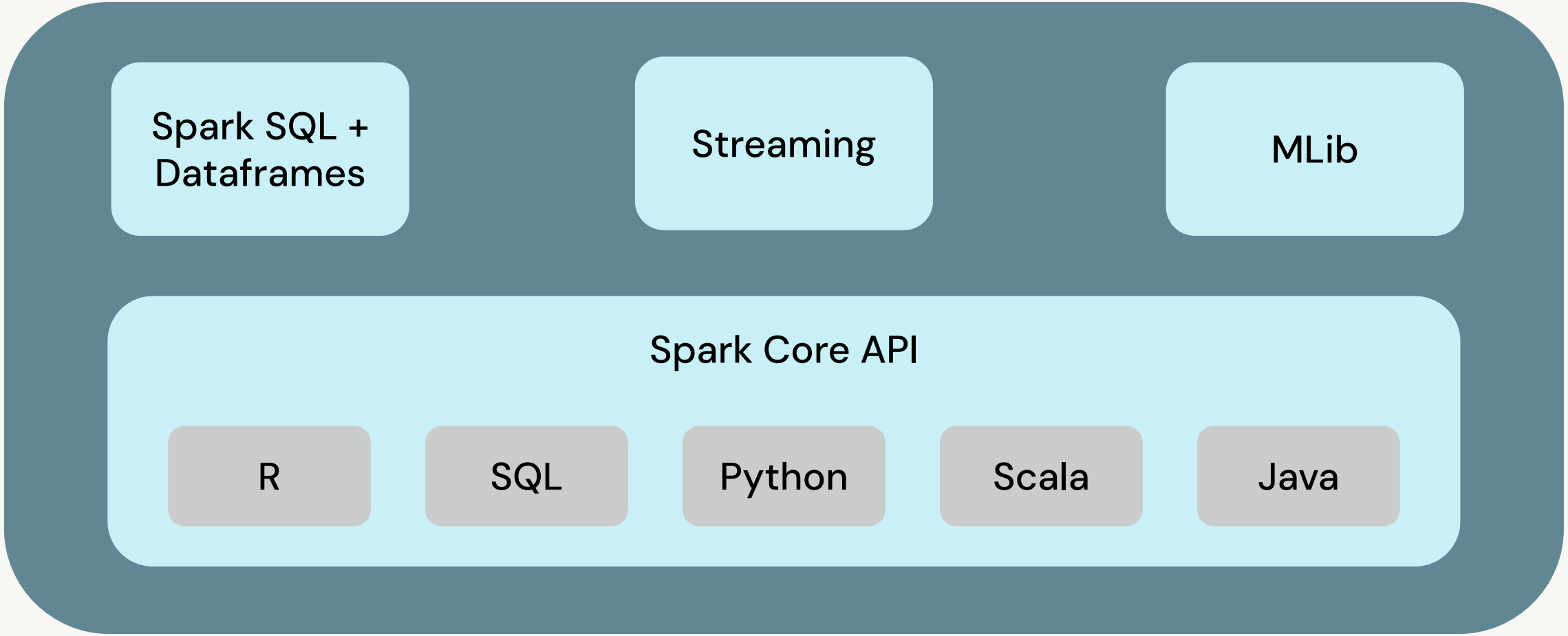
Rápido

Fácil de usar

Unificado

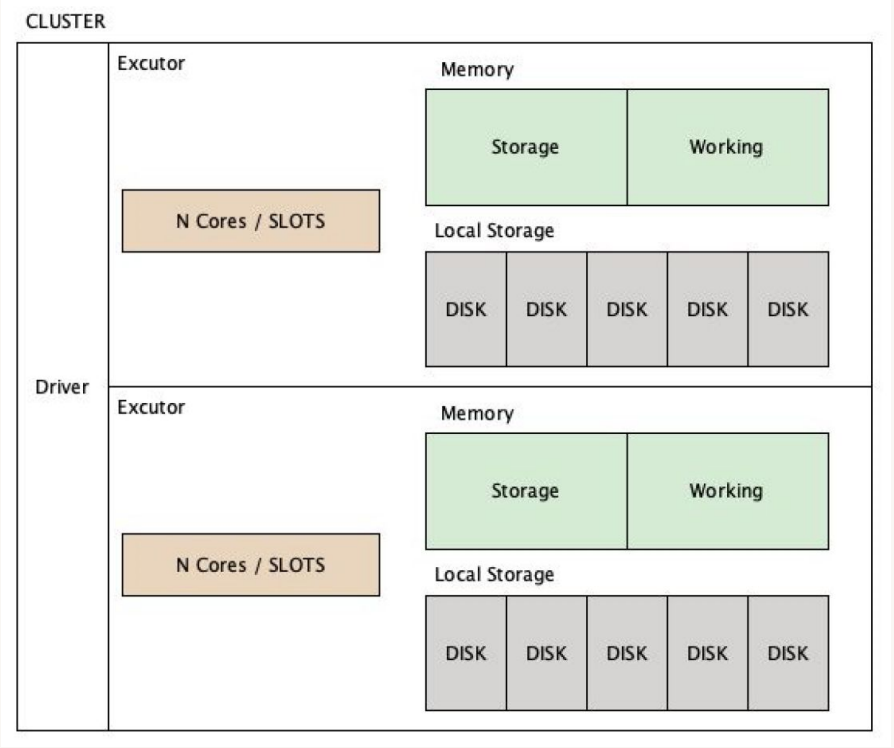
Spark Overview

As APIs do Spark



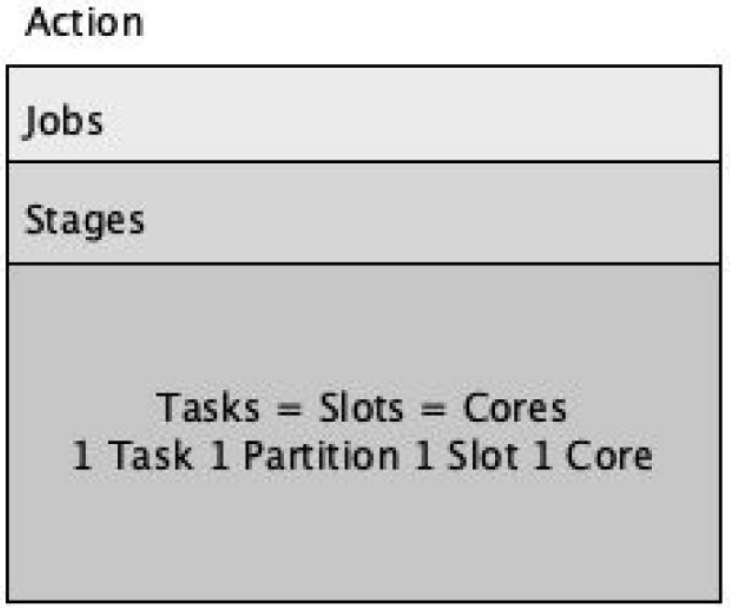
Spark Introduction – Hardware

- Hardware – how does Spark run on your cluster?
  - *Assumptions*
    - Single driver
    - 1 : n many executors
  - One executor per server (this could be 1:n)
    - Contains all cores, memory, disk, and network that can be used to run Spark tasks
  - Memory – *50% for each by default*
    - Storage - Used for caching objects
    - Working - Used for transformations

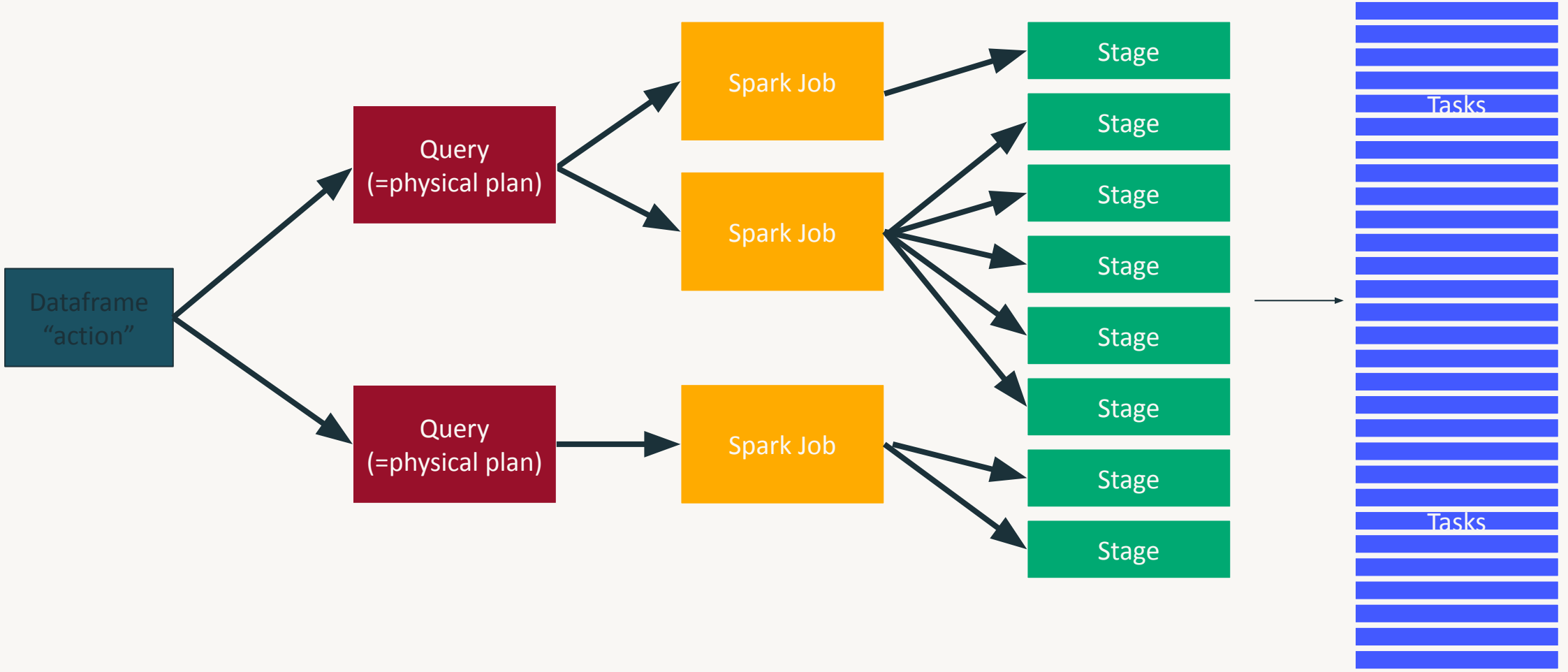


Spark Introduction - Software

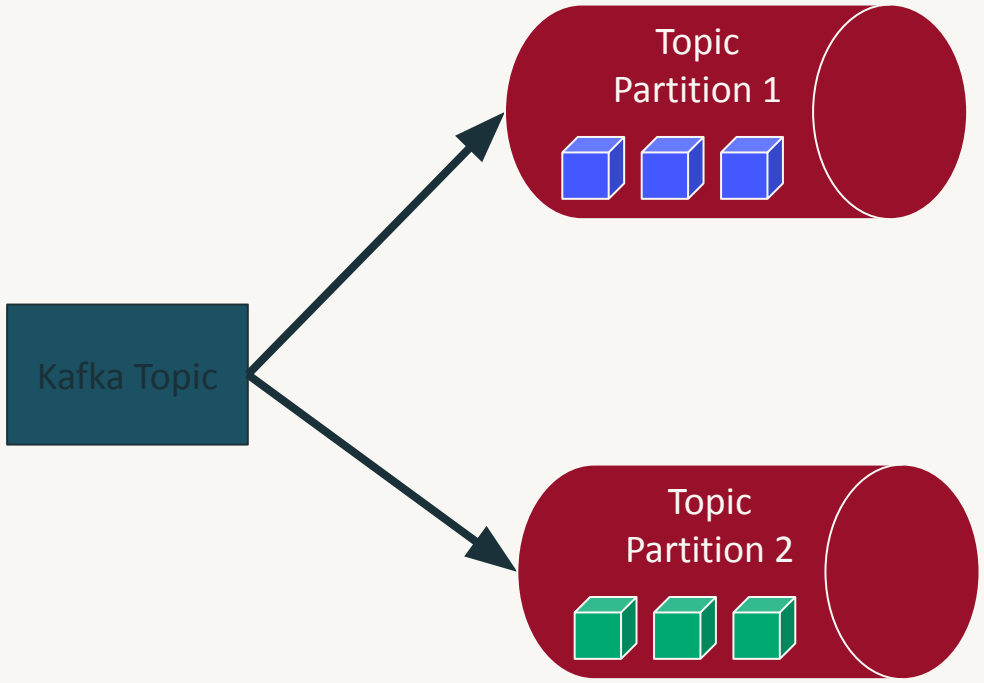
- Software - how does Spark run on your cluster?
  - Spark operates on your data via Tasks - aka Slots, Cores, Partitions
    - **Base unit of parallelism - execute same set of operations on partitions in parallel**
  - Stages - Organized lineage of transformations - transforms structure/contents of data to desired output.
    - Narrow - Does not require a shuffle.
    - *filter, union, coalesce*
    - Wide - Requires a shuffle, needs key-grouping.
    - *groupBy, join, dropDuplicates*
  - Jobs - Determined by actions in code.
  - Actions - Eager operations that kick off n number of Jobs
    - *cache, creating views, columns, explain, writing*



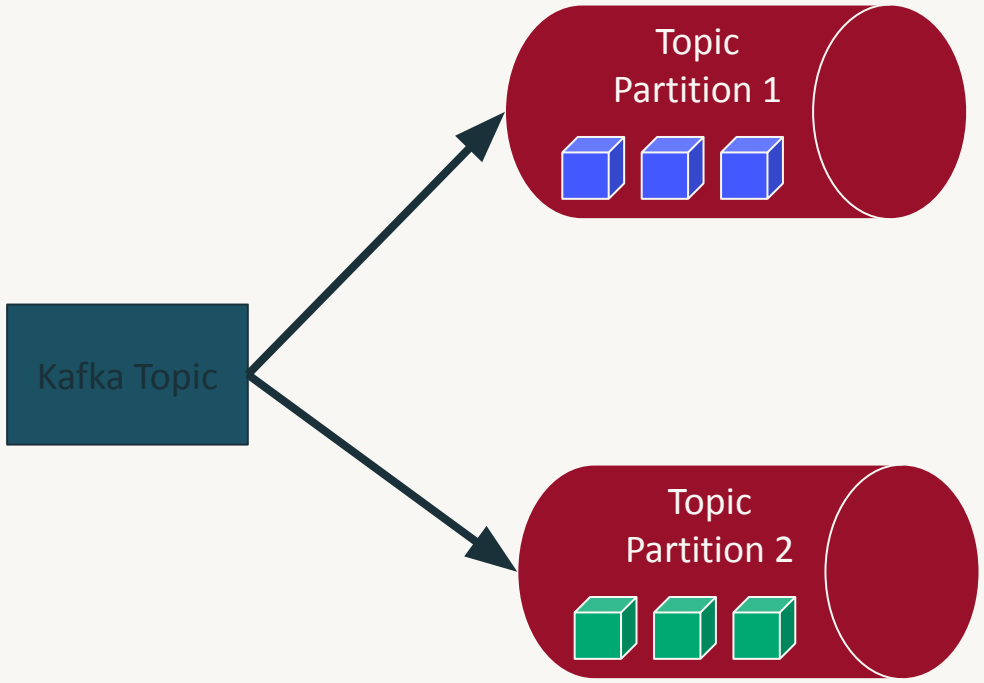
Spark Introduction - Software

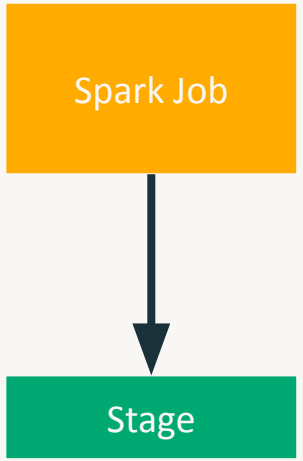


Kafka – Distributed / Partitioned Queue



Kafka – Distributed / Partitioned Queue

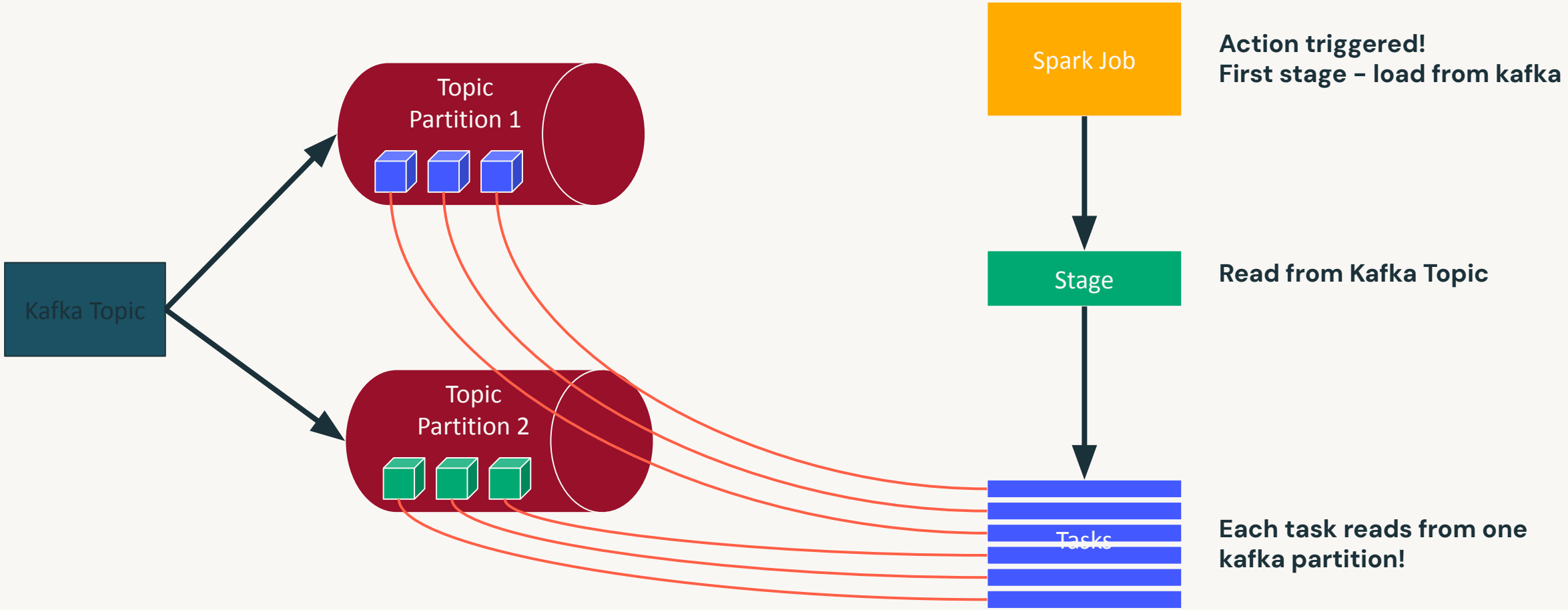




Action triggered!
First stage – load from kafka

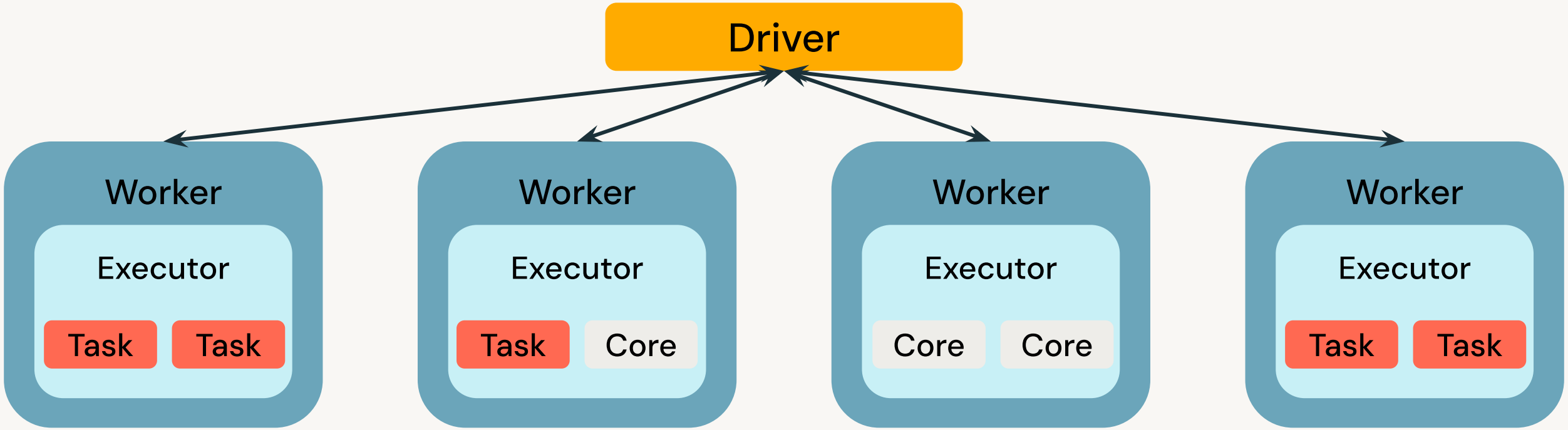
Read from Kafka Topic

Kafka – Distributed / Partitioned Queue



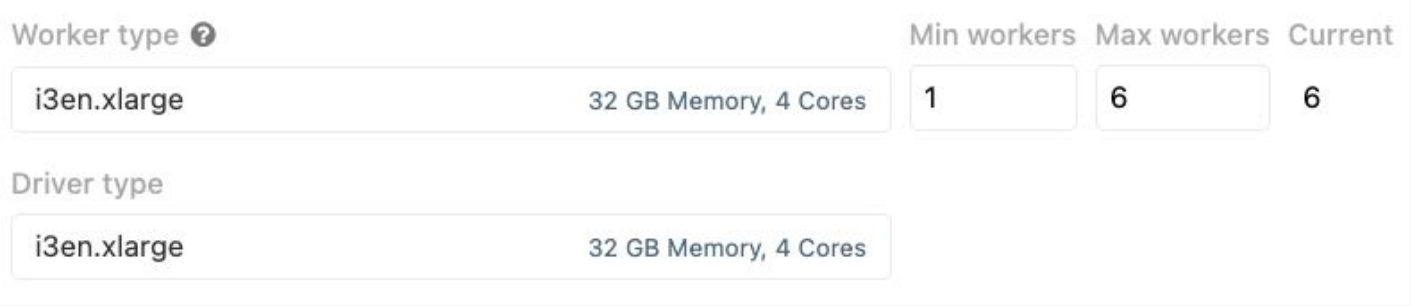
Spark Overview

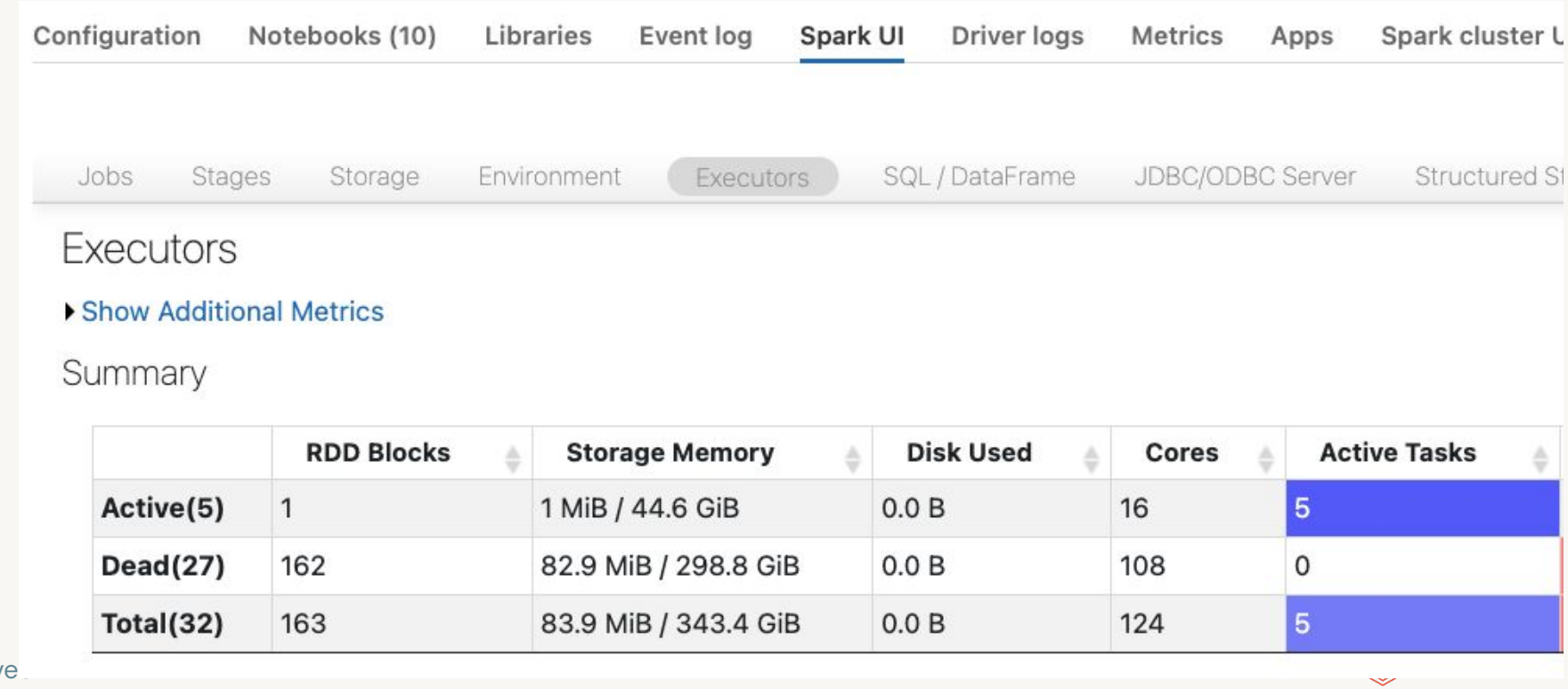
Cluster de Spark (1/2)



Spark Overview

Cluster de Spark (2/2)





	RDD Blocks	Storage Memory	Disk Used	Cores	Active Tasks
Active(5)	1	1 MiB / 44.6 GiB	0.0 B	16	5
Dead(27)	162	82.9 MiB / 298.8 GiB	0.0 B	108	0
Total(32)	163	83.9 MiB / 343.4 GiB	0.0 B	124	5

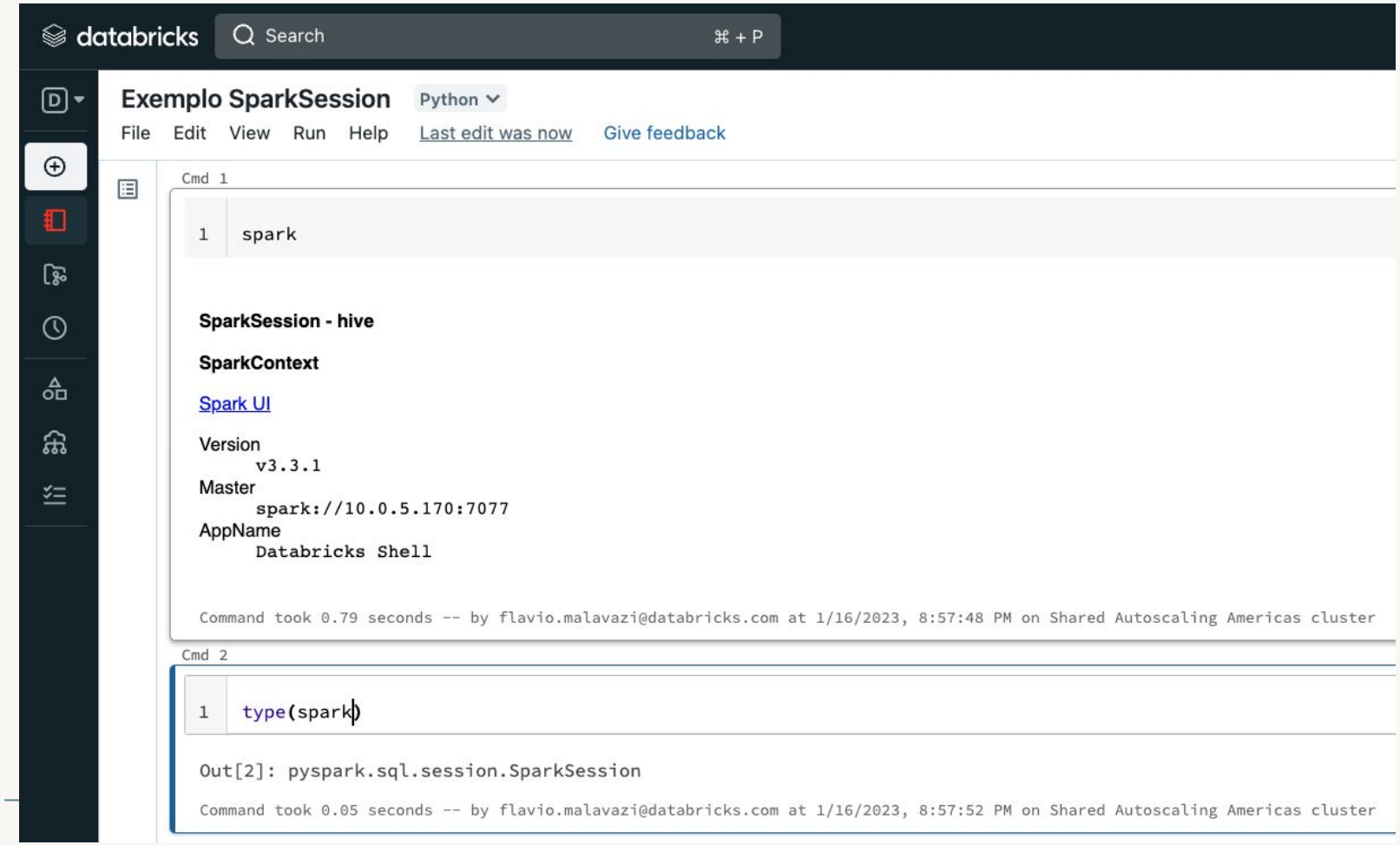
Spark Overview

Spark Session

SparkSession

A SparkSession é seu ponto de entrada para todas as funcionalidades da API de DataFrames

Nos notebooks Databricks, ela é automaticamente criada para você



SparkSession

A SparkSession tem métodos muito úteis

spark.sql	Retorna um DataFrame representando o resultado de uma query em SQL
spark.table	Retorna a tabela especificada como um DataFrame
spark.read	Retorna um DataFrameReader que pode ser utilizado para ler dados em um DataFrame
spark.range	Cria um DataFrame com uma coluna contendo elementos em um range do começo ao fim (exclusivo) com um valor de degrau e um número de partições
spark.createDataFrame	Cria um DataFrame a partir de uma lista de tuplas, usado principalmente para testes

Spark Overview

SparkSQL

SparkSQL

Processando dados estruturados em Spark

O módulo SparkSQL permite a utilização de diversas interfaces, dentre elas: SQL, Python, Scala, R e Java

A mesma tarefa pode ser executada de diferentes maneiras...

SQL

```
SELECT id, result
FROM exams
WHERE result > 70
ORDER BY result
```

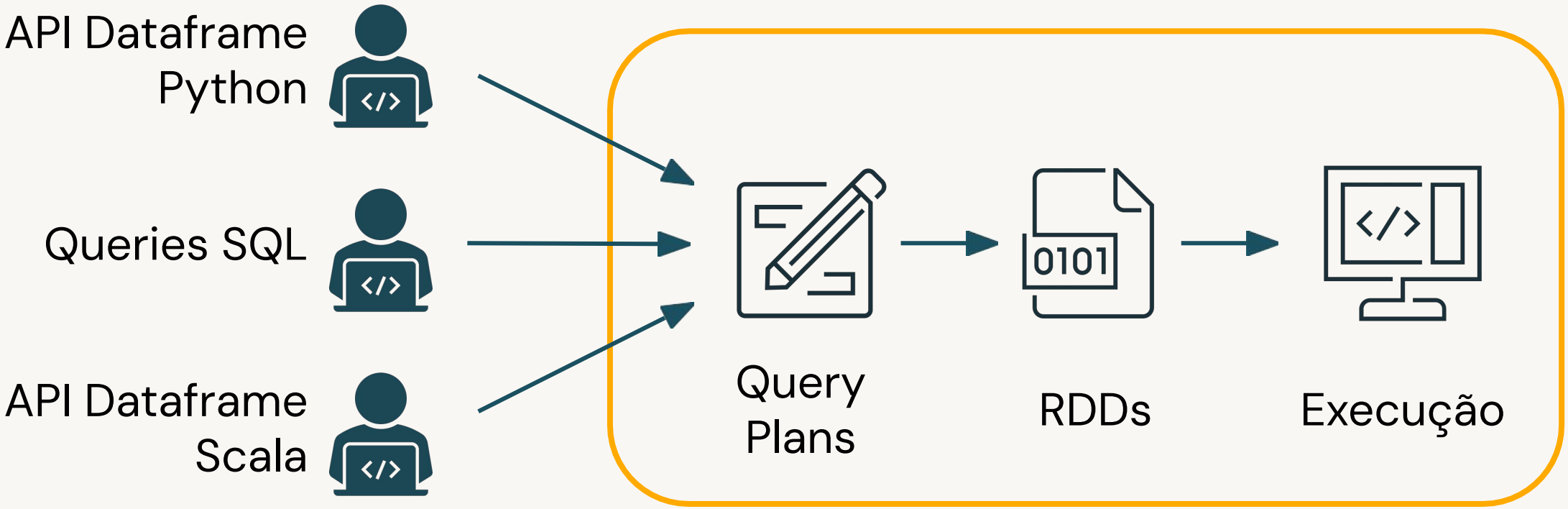
DataFrame API

Python, Scala, Java, R

```
spark.table("exams")
  .select("id", "result")
  .where("result > 70")
  .orderBy("result")
```

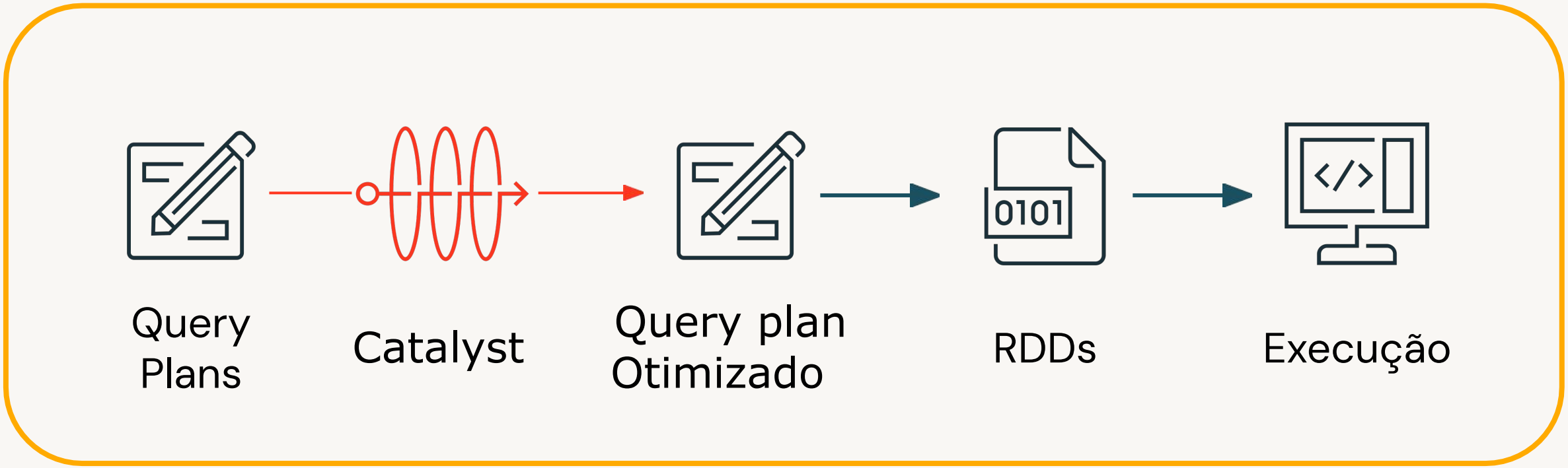
SparkSQL

O SparkSQL executa todas as queries usando o mesmo motor



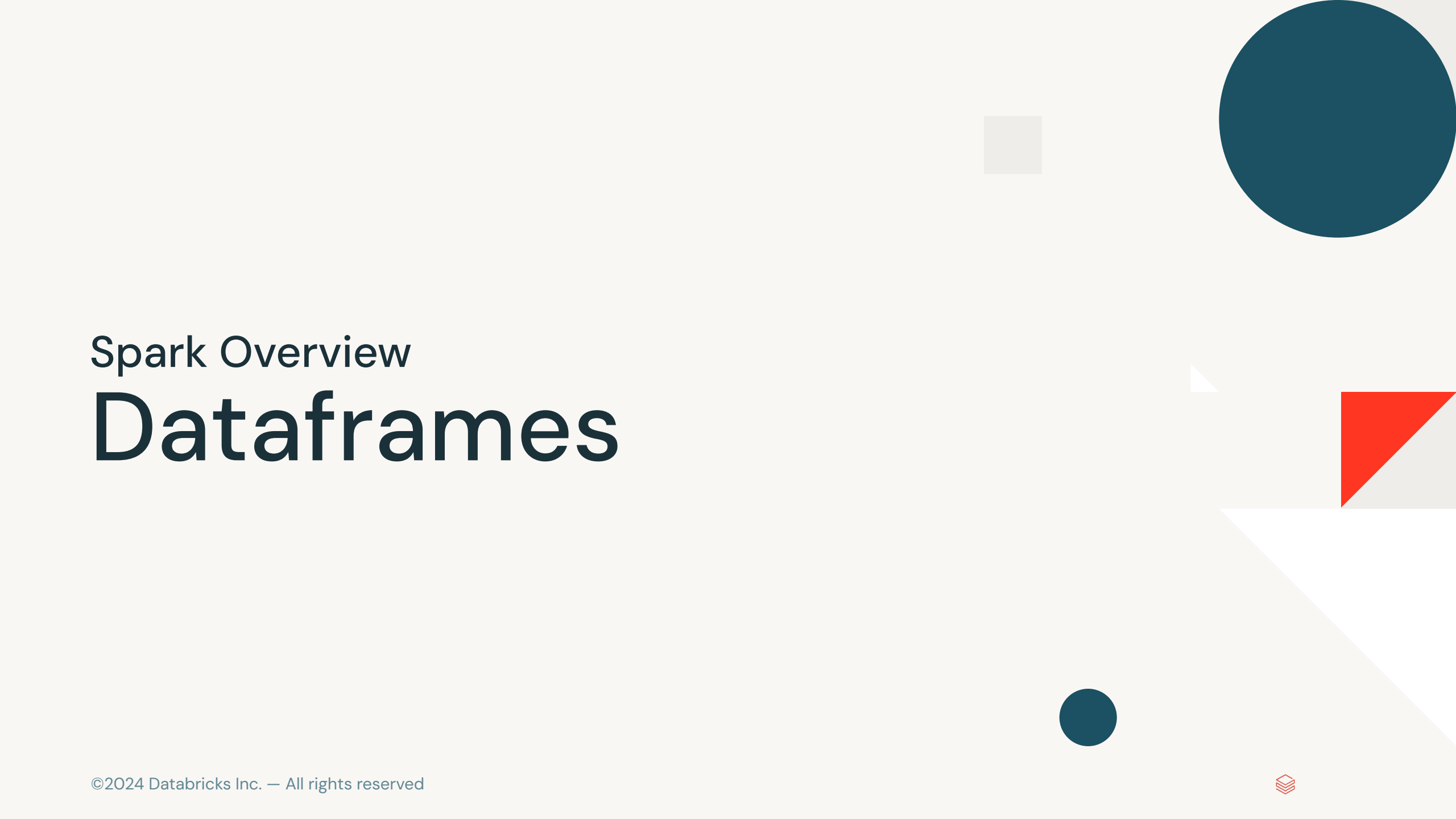
SparkSQL

O Catalyst cuida de otimizar as suas queries antes da execução em um cluster Spark



Spark Overview

Dataframes



Dataframes são coleções distribuídas de dados agrupadas em colunas nomeadas

item_id	name	price	qty
M_PREM_Q	Premium Queen Mattress	1795	35
M_STAN_F	Standard Full Mattress	945	24
M_PREM_F	Premium Full Mattress	1695	45
M_PREM_T	Premium Twin Mattress	1095	18

Dataframes

Um schema define os nomes e tipos das colunas em um DataFrame

DataFrame

item_id	name	price	qty
M_PREM_Q	Premium Queen Mattress	1795	35
M_STAN_F	Standard Full Mattress	945	24
M_PREM_F	Premium Full Mattress	1695	45
M_PREM_T	Premium Twin Mattress	1095	18

Schema

```
item_id: string  name: string  price: double  qty: int
```

Dataframes

Transformações são métodos que retornam um novo DF, e são avaliados "preguiçosamente"

```
df.select("id", "result")
    .where("result > 70")
    .orderBy("result")
```

Dataframes

Ações são métodos que disparam a computação de resultados

```
df.count()

df.collect()

df.show()
```

Dataframes

Uma ação é necessária para disparar a execução de qualquer transformação de DF

```
df.select("id", "result")
  .where("result > 70")
  .orderBy("result")
```

Spark Overview

Funções Comuns

Funções comuns

Operadores e métodos de colunas

&&, || / &, |	E, OU booleano em Scala / Python
***, + , <, >=**	Matemática e operadores de comparação
===, =!= / ==, !=	Testes de igualdade ou desigualdade em Scala / Python
alias, as	Utiliza um apelido para colunas colunas, o `**as**` funciona apenas em Scala
cast, astype	Faz o cast de uma coluna para um tipo diferente, o `astype` só funciona em Python
isNull, isNotNull, isNan	Teste de nulo, não nulo ou é NaN
asc, desc	Retorna uma expressão de ordenação baseada em uma ordem crescente/decrescente de uma coluna.

Funções comuns

Métodos de transformação de DataFrames

select	Retorna um novo DataFrame após computar uma expressão dada para cada elemento
drop	Retorna um novo DataFrame menos uma coluna
withColumnRenamed	Retorna um novo DataFrame com uma coluna renomeada
withColumn	Retorna um novo DataFrame adicionando uma coluna ou substituindo uma coluna existente que tenha o mesmo nome
filter, where	Filtra linhas usando a condição dada
sort, orderBy	Retorna um novo DataFrame ordenado pela expressão dada
dropDuplicates, distinct	Retorna um novo DataFrame sem linhas duplicadas
limit	Retorna um novo DataFrame limitado às primeiras n linhas
groupBy	Agrupa o DataFrame usando as colunas específicadas, para que se possam executar agregações nelas

Funções comuns

Ações de DataFrames

show	Exibe as n primeiras linhas de um DataFrame em formato tabular
count	Retorna o número de linhas do DataFrame
describe, summary	Computa estatísticas básicas para colunas numéricas e de strings
first, head	Retorna a primeira linha
collect	Retorna um array que contém todas as linhas do DataFrame
take	Retorna um array com as primeiras n linhas do DataFrame

Funções comuns

Métodos de linha (Python)

index	Retorna o primeiro índice de um valor
count	Retorna o número de ocorrências de um valor
asDict	Retorna como dicionário (dict)
row.key	Acessa campos como atributos
row["key"]	Acessa campos como valores em um dicionário
key in row	Procura dentre as chaves da linha

Funções comuns

Métodos de linha (Scala)

fieldIndex	Retorna o índice de um campo nomeado
get(i)	Retorna o valor na posição i com tipo Any
getAs[T]	Retorna o valor de um dado campo `fieldName`
getDouble	Retorna o valor na posição i como um double primitivo
length	Número de elementos na linha
mkString	Exibe todos os elementos desta sequência em uma string
isNullAt	Checa se o valor na posição i é nulo

Delta Fundamentals

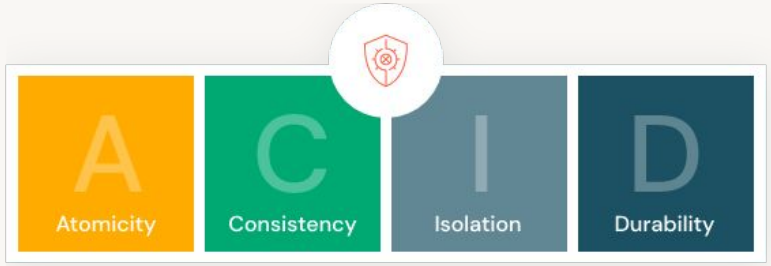
Uma única fonte de verdade para todos os seus dados

Um novo padrão para a criação de data lakes

- O Databricks SQL armazena e processa dados usando o formato Delta Lake baseado em Parquet **de software livre**
- Delta Lake agrega **qualidade**, **confiabilidade** e **desempenho** para seus data lakes já existentes
- Fornece uma **estrutura comum de gerenciamento de dados** para Batch & Streaming, ETL, Analytics & ML

DELTA LAKE + THE LINUX FOUNDATION

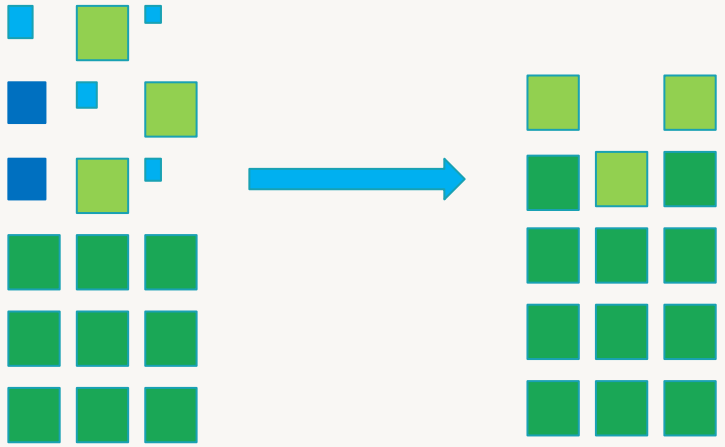
- ✓ Transações ACID
- ✓ Viagem no tempo
- ✓ Imposição de esquema
- ✓ Colunas de identidade
- ✓ Indexação avançada
- ✓ Cache
- ✓ Ajuste automático
- ✓ Controles de acesso refinados e baseados em funções
- ✓ Suporte a Python, SQL, R, Scala



Evitando o problema dos small files

O Databricks gerencia este desafio automaticamente em runtimes mais recentes e tabelas do Unity Catalog

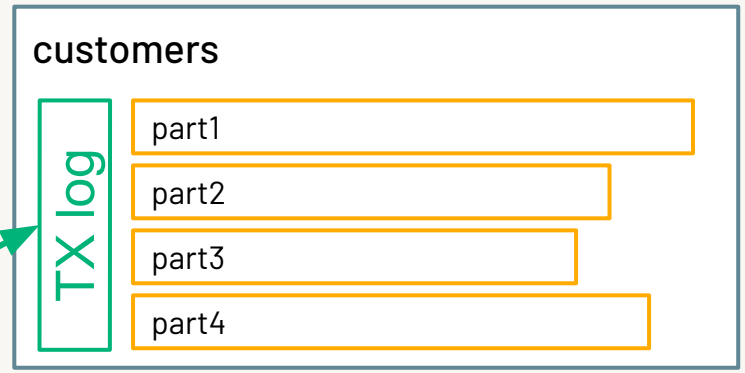
- Muitos arquivos pequenos aumenta o overhead de leituras
- Poucos arquivos muito grandes reduz o paralelismo nas leituras
- Over-partitioning é um problema comum
- O Databricks ajusta automaticamente o tamanho do arquivo para tabelas do Delta Lake
- O Databricks compacta pequenos arquivos automaticamente na escrita com o auto-optimize



Delta Tables

O log de transações fornece uma camada de metadados para consistência

Dados em Parquet




```
%fs ls /tmp/bernhard/loan_by_state_delta
```

path
dbfs:/tmp/bernhard/loan_by_state_delta/_delta_log/
dbfs:/tmp/bernhard/loan_by_state_delta/part-00000-cd80fd12-457b-4058-a904-3628c6e90a57-c000.snappy.parquet
dbfs:/tmp/bernhard/loan_by_state_delta/part-00004-69b2735a-18d1-474e-8318-ecd13ca410a7-c000.snappy.parquet
dbfs:/tmp/bernhard/loan_by_state_delta/part-00009-05545d85-f051-4a37-852a-18298fb4f3cd-c000.snappy.parquet
dbfs:/tmp/bernhard/loan_by_state_delta/part-00010-3fa88ba6-61cc-45cd-8c0d-e0949d1bbcce-c000.snappy.parquet

https://databricks.com/blog/2019/08/21/diving-into-delta-lake-unpacking-the-transaction-log.html

1. Dados difíceis de anexar (append)
2. Modificação de dados existentes difícil
3. Jobs falhando no meio do caminho
4. Operações em tempo real difíceis
5. Caro manter versões de dados históricos
6. Difícil de lidar com metadados grandes
7. Problemas de "muitos arquivos"
8. Baixo desempenho
9. Problemas de qualidade de dados

Transações ACID

Torne cada operação transacional

- Ele é totalmente bem-sucedido - ou é totalmente abortado para tentativas posteriores

```
/caminho/para/tabela/_delta_log
  -  0000.json
  -  0001.json
  -  0002.json
  -  ...
  -  0010.parquet
```

1. Dados difíceis de anexar (append)
2. Modificação de dados existentes difícil
3. Jobs falhando no meio do caminho
4. Operações em tempo real difíceis
5. Caro manter versões de dados históricos
6. Difícil de lidar com metadados grandes
7. Problemas de "muitos arquivos"
8. Baixo desempenho
9. Problemas de qualidade de dados

Transações ACID

Torne cada operação transacional

- Ele é totalmente bem-sucedido - ou é totalmente abortado para tentativas posteriores

```
/caminho/para/tabela/_delta_l
og
  -  0000.json
  -  0001.json
  -  0002.json
  -  ...
  -  0010.parquet
```

Adicionar arquivo1.parquet
Adicionar file2.parquet
...

1. Dados difíceis de anexar (append)
2. Modificação de dados existentes difícil
3. Jobs falhando no meio do caminho
4. Operações em tempo real difíceis
5. Caro manter versões de dados históricos
6. Difícil de lidar com metadados grandes
7. Problemas de "muitos arquivos"
8. Baixo desempenho
9. Problemas de qualidade de dados

Transações ACID

Torne cada operação transacional

- Ele é totalmente bem-sucedido - ou é totalmente abortado para tentativas posteriores

```
/caminho/para/tabela/_delta_log
 -  0000.json
 -  0001.json
 -  0002.json
 -  ...
 -  0010.parquet
```

Remover arquivo1.parquet
Adicionar arquivo3.parquet

1. Dados difíceis de anexar (append)
2. Modificação de dados existentes difícil
3. Jobs falhando no meio do caminho
4. Operações em tempo real difíceis
5. Caro manter versões de dados históricos
6. Difícil de lidar com metadados grandes
7. Problemas de "muitos arquivos"
8. Baixo desempenho
9. Problemas de qualidade de dados

Transações ACID

Torne cada operação transacional

- Ele é totalmente bem-sucedido - ou é totalmente abortado para tentativas posteriores

```
/caminho/para/tabela/_delta_log
 -  0000.json
 -  0001.json
 -  0002.json
 -  ...
 -  0010.parquet
 -  0010.json
 -  0011.json
```

1. Dados difíceis de anexar (append)
2. Modificação de dados existentes difícil
3. Jobs falhando no meio do caminho
4. Operações em tempo real difíceis
5. Caro manter versões de dados históricos
6. Difícil de lidar com metadados grandes
7. Problemas de "muitos arquivos"
8. Baixo desempenho
9. Problemas de qualidade de dados

Transações ACID

Torne cada operação transacional

- Ele é totalmente bem-sucedido - ou é totalmente abortado para tentativas posteriores

Revisar transações anteriores

- Todas as transações são registradas e você pode voltar no tempo para revisar versões anteriores dos dados (ou seja, *time travel*)

```
SELECT * FROM eventos
CARIMBO DE DATA/HORA A
PARTIR DE ...

SELECT * FROM eventos
VERSÃO A PARTIR DE ...
```

1. Dados difíceis de anexar (append)
2. Modificação de dados existentes difícil
3. Jobs falhando no meio do caminho
4. Operações em tempo real difíceis
5. Caro manter versões de dados históricos
6. Difícil de lidar com metadados grandes
7. Problemas de "muitos arquivos"
8. Baixo desempenho
9. Problemas de qualidade de dados

Spark por debaixo dos panos

- O Spark foi criado para lidar com grandes quantidades de dados
- Todos os metadados do Delta Lake armazenados no formato Parquet aberto
- Partes dele armazenadas em cache e otimizadas para acesso rápido
- Dados e seus metadados sempre coexistem.
  Não há necessidade de manter os dados <>do catálogo sincronizados

1. Dados difíceis de anexar (append)
2. Modificação de dados existentes difícil
3. Jobs falhando no meio do caminho
4. Operações em tempo real difíceis
5. Caro manter versões de dados históricos
6. Difícil de lidar com metadados grandes
7. Problemas de "muitos arquivos"
8. Baixo desempenho
9. Problemas de qualidade de dados

Indexação

Otimize automaticamente um layout que permite acesso rápido

- Particionamento: layout para consultas típicas
- Ignorar dados: remover arquivos com base em estatísticas numéricas
- Ordenação Z: layout para otimizar várias colunas

```
Eventos OPTIMIZE
ZORDER BY (eventType)
```

1. Dados difíceis de anexar (append)
2. Modificação de dados existentes difícil
3. Jobs falhando no meio do caminho
4. Operações em tempo real difíceis
5. Caro manter versões de dados históricos
6. Difícil de lidar com metadados grandes
7. Problemas de "muitos arquivos"
8. Baixo desempenho
9. Problemas de qualidade de dados

Validação e expectativas de esquema

Validação e evolução do esquema

- Todos os dados nas tabelas delta devem aderir a um esquema estrito
- Inclui evolução de esquema em operações de mesclagem

```
MERGE INTO events
USING changes
ON events.id = changes.id
WHEN MATCHED THEN
  UPDATE SET *
WHEN NOT MATCHED THEN
  INSERT *
```

Delta Fundamentals

Time Travel

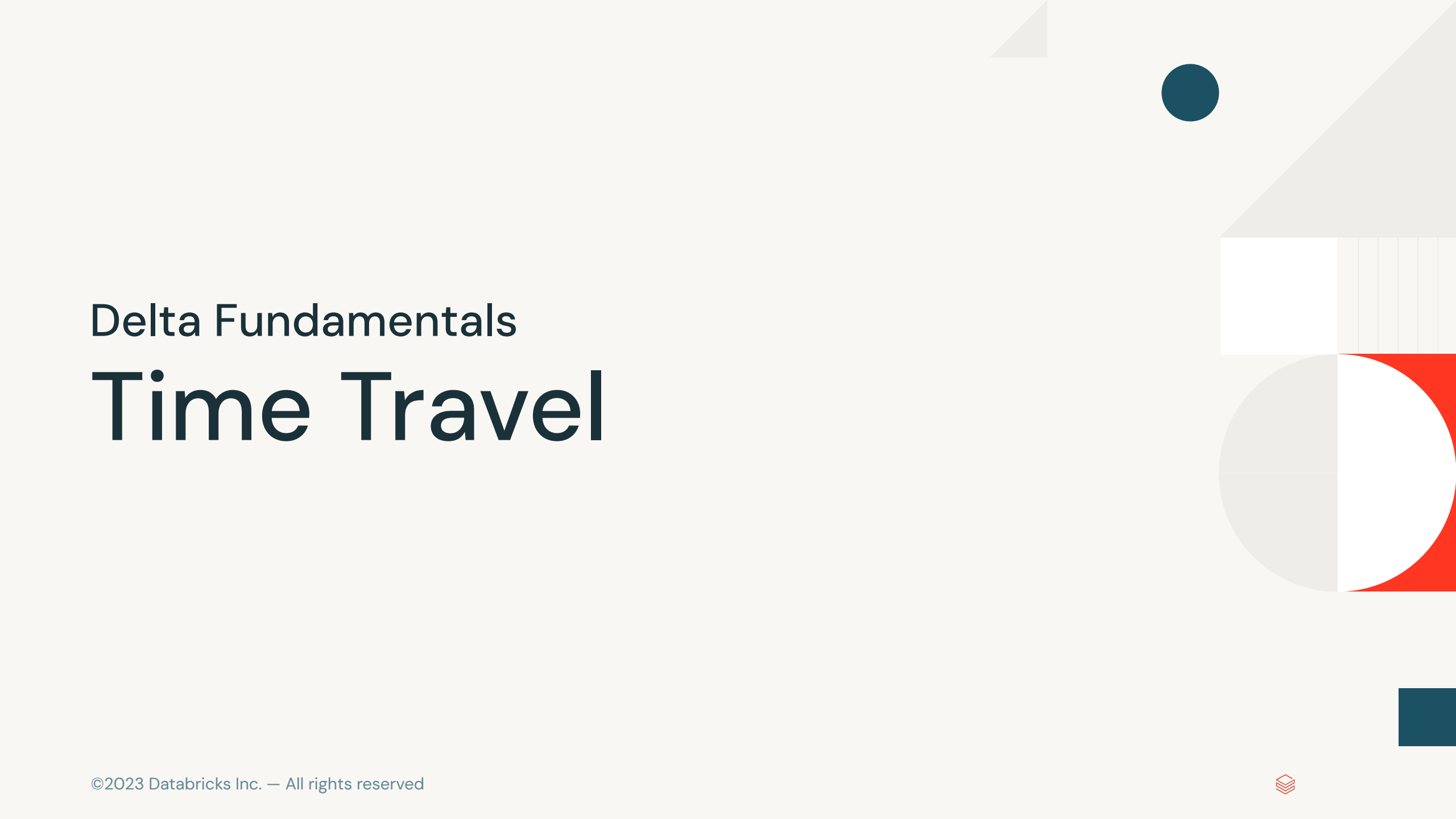


Table history

Características principais

- Um "registro de auditoria" das alterações em seus dados
- Veja quem mudou o quê, quando

Sintaxe

```
DESCRIBE HISTORY customers -- Delta table
DESCRIBE HISTORY delta.`path/to/table`
```

Time Travel

Reproduza experimentos e relatórios

```
SELECT count(*) FROM events
TIMESTAMP AS OF timestamp;

SELECT count(*) FROM events
VERSION AS OF version;
```

```
spark.read.format("delta").option("timestampAsOf",
timestamp_string).table("events")
```

Reversão de gravações incorretas acidentais

```
RESTORE TABLE events
    TO TIMESTAMP AS OF 2020-01-18;

RESTORE TABLE events
    TO VERSION AS OF 123;
```

Delta Fundamentals

Schema Enforcement Evolution (drift) e Validation

Schema Enforcement e Evolution

- **Schema enforcement**
  - Default

- **Schema evolution (drift) – multiple flavors**
  - Acrescentar um DataFrame com um esquema em evolução a uma tabela Delta
  - Mesclar um DataFrame com um esquema em evolução para uma tabela Delta
  - Substituir uma tabela Delta por dados com um esquema em evolução

Schema evolution

Acrescentar um DataFrame com esquema em evolução a uma tabela Delta

- **Caso de uso simples para ilustração**
  - Existe uma tabela Delta existente com 2 colunas - `id` e `name`.
  - O DataFrame mais recente tem 3 colunas - `id`, `name` e `year`.
    - Alteração: uma nova coluna foi adicionada ao conjunto de dados: `ano`.
  - O requisito é ***acrescentar*** os novos dados à tabela Delta existente e também manter a nova coluna na tabela Delta.

```
df.write
   .format("delta")
   .mode("append")
   .option("mergeSchema", "true")
   .saveAsTable("my_delta_table")
```

Schema evolution

Mesclar um DataFrame com esquema em evolução para uma tabela Delta

- **Caso de uso simples para ilustração**
  - Existe uma tabela Delta existente com 2 colunas - `id` e `name`.
  - O DataFrame mais recente tem 3 colunas - `id`, `name` e `year`.
    - Alteração: uma nova coluna foi adicionada ao conjunto de dados: `ano`.
  - O requisito é mesclar os novos dados com a tabela Delta existente e também manter a nova coluna na tabela Delta.

```
spark.conf.set("spark.databricks.delta.schema.autoMerge.enabled", "true")
```

```
MERGE INTO merge_target_table target    -- Delta table
USING source_table_for_merge source
ON target.id = source.id
WHEN MATCHED
     THEN UPDATE SET *
WHEN NOT MATCHED
     THEN INSERT *
```

Schema evolution

Substituir uma tabela Delta por um novo DataFrame com esquema em evolução

- **Caso de uso simples para ilustração**
  - Existe uma tabela Delta existente com 3 colunas - `id`, `nome` e `ano`.
  - O DataFrame mais recente tem 2 colunas - `id` e `tech`.
    - Uma coluna foi renomeada: `nome` para `tecnologia`.
    - Outra coluna foi descartada: `ano`
  - O requisito é gravar esses dados na mesma tabela e ***substituir os dados e o esquema*** da tabela Delta existente.

```
df.write
  .format("delta")
  .mode("overwrite")
  .option("overwriteSchema", "true")
  .saveAsTable("my_delta_table")
```

NOT NULL Constraint

```
CREATE TABLE events (
     id LONG NOT NULL,
    date STRING NOT NULL,
    location STRING,
    description STRING);
```

```
ALTER TABLE events_1 CHANGE COLUMN date DROP NOT NULL;
ALTER TABLE events_1 CHANGE COLUMN id   SET  NOT NULL;
```

Características principais

- As restrições são definidas no nível do esquema de tabela
- Você pode criar ou descartar restrições NOT NULL usando o comando ALTER TABLE CHANGE COLUMN

CHECK Constraint

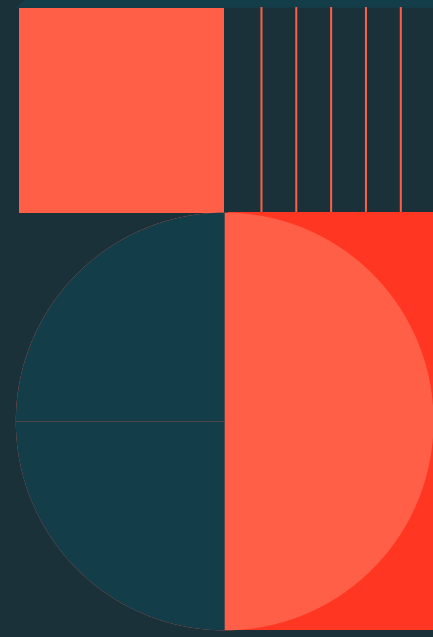
```
CREATE TABLE events (
     id LONG NOT NULL,
    date STRING NOT NULL,
    location STRING,
    description STRING);
```

```
ALTER TABLE events_2 ADD CONSTRAINT  dateWithinRange CHECK date > '1900-01-01';
ALTER TABLE events_2 DROP CONSTRAINT dateWithinRange;
```

Características principais

- ALTER TABLE ADD CONSTRAINT verifica se todas as linhas existentes atendem à restrição antes de adicioná-la à tabela

Demo
00 - Databricks Fundamentals

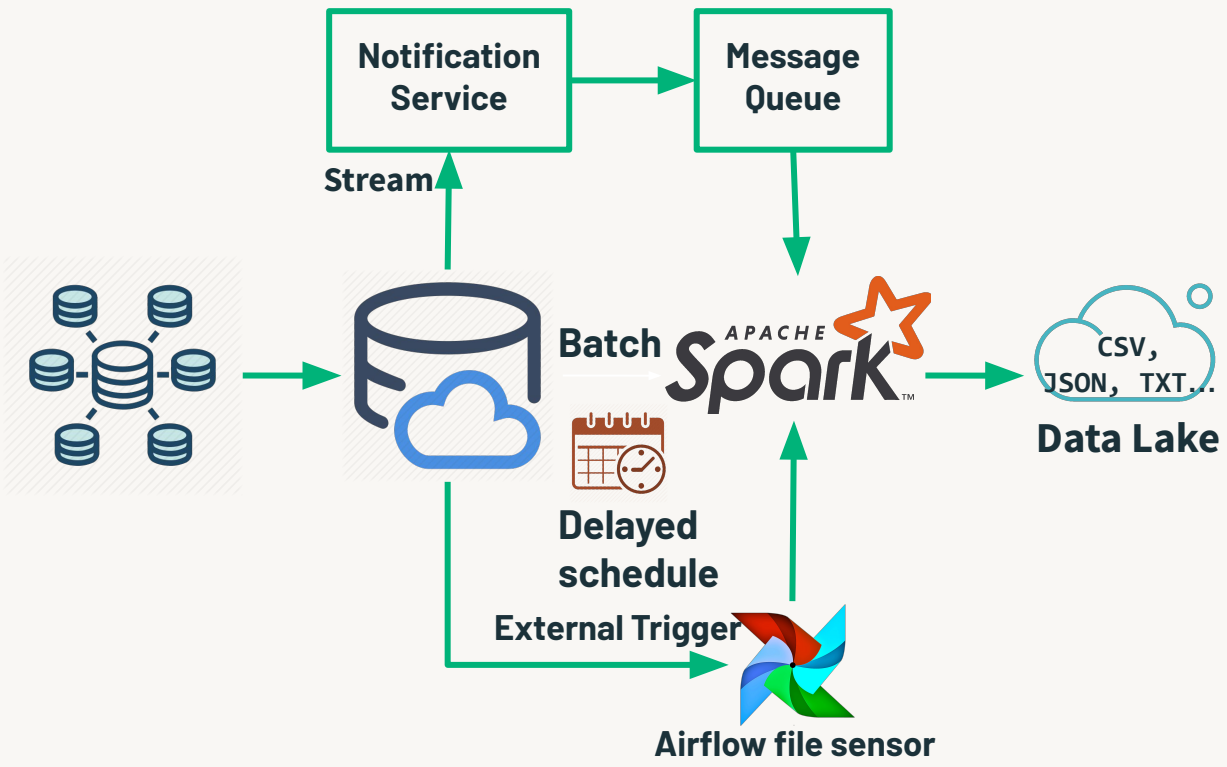


Auto Loader

Databricks Ingest: Auto Loader

Load new data easily and efficiently as it arrives in cloud storage

Before



Gets too complicated for multiple jobs

After

- Pipe data from cloud storage into your data lake with Delta Lake as it arrives
- "Set and forget" model eliminates complex setup

Launch Blog Post

What Autoloader Does For You

Auto-Ingest New Files

Incrementally **process new files** as they land in cloud object storage without costly file listing procedures or state information handling

Scalable Framework

Auto Loader configures cloud notification and message queues to ensure that all files are sent to be ingested as they land and can scale to **millions of file** per directory

Infer & Evolve Schemas

Identify schema on initialization and **detect changes** over time from structured sources and semi-structured data formats

Add hints for enforcement where some schema is known

Easy to Get Started

Easy to deploy and **easy to use** with automated deployment of services making Auto Loader nearly set and forget

Scalable exactly-once data ingestion with Auto Loader

```
df = spark
       .readStream
       .format("cloudFiles")
       .option("cloudFiles.format", "json")
       .load("abfss://..." or "s3://")
       .<apply your transformations>
       .writeStream
       .option("checkpointLocation","/chk/path")
       .start("/out/path")
```

- Incrementally and efficiently process new data files as they arrive in cloud storage
  - **File Notification mode** enables **event-driven** ingestion (automatically sets up Event Grid / Amazon SNS + Azure Queue Storage / Amazon SQS on your behalf)
- Automatically **infer schema** of incoming files or superimpose what you know with **Schema Hints**
- Automatic **schema evolution**
- **Rescue data column** – never lose data again

Schema Evolution	JSON	CSV	AVRO	PARQUET
	✅	✅	✅	✅

Streaming Loads with Auto Loader

```
df = spark.readStream.format("cloudFiles")
    .option("cloudFiles.format", "json")
    .schema(schema)
    .load("/input/path")


df.writeStream
  .format("delta")
  .option("checkpointLocation", "/checkpoint/path")
  .start("/out/path")
```

Batch Loads with Auto Loader

```
df = spark.readStream.format("cloudFiles")
  .option("cloudFiles.format", "json")
  .schema(schema)
  .load("/input/path")


df.writeStream
  .format("delta")
  .trigger(Trigger.Once)
  .option("checkpointLocation", "/checkpoint/path")
  .start("/out/path")
```

Demo

01 - Bronze Layer (autoloader - csv)

Lab

01Lab - Bronze Layer (autoloader - json) 15min

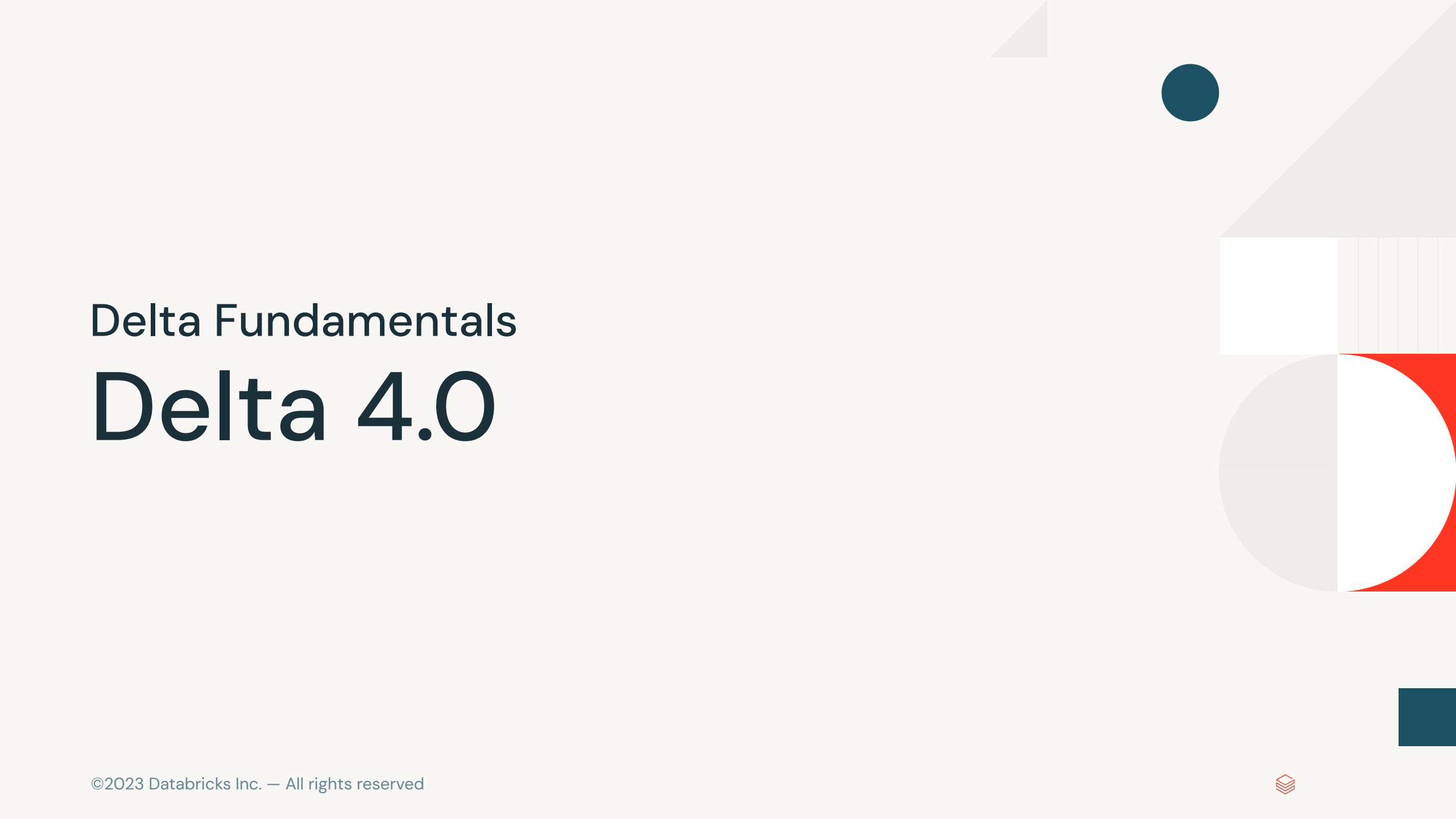
Demo

02 - Silver Layer

03 - Gold Layer (Join)

Delta Fundamentals

Delta 4.0



Delta Lake:
O formato aberto de lakehouse mais adotado

Escalável	Popular	Dominante	Confiável	Inovador	Aberto
4+ exabytes Processados por dia	**1B+** Clusters por ano	**60%+** Adoção no Fortune 500	**>10K+** Empresas em produção	**80+** Features novas / ano	**>500** Contribuidores

2x crescimento anual

O Maior Delta Lake release até agora

Delta 3.x

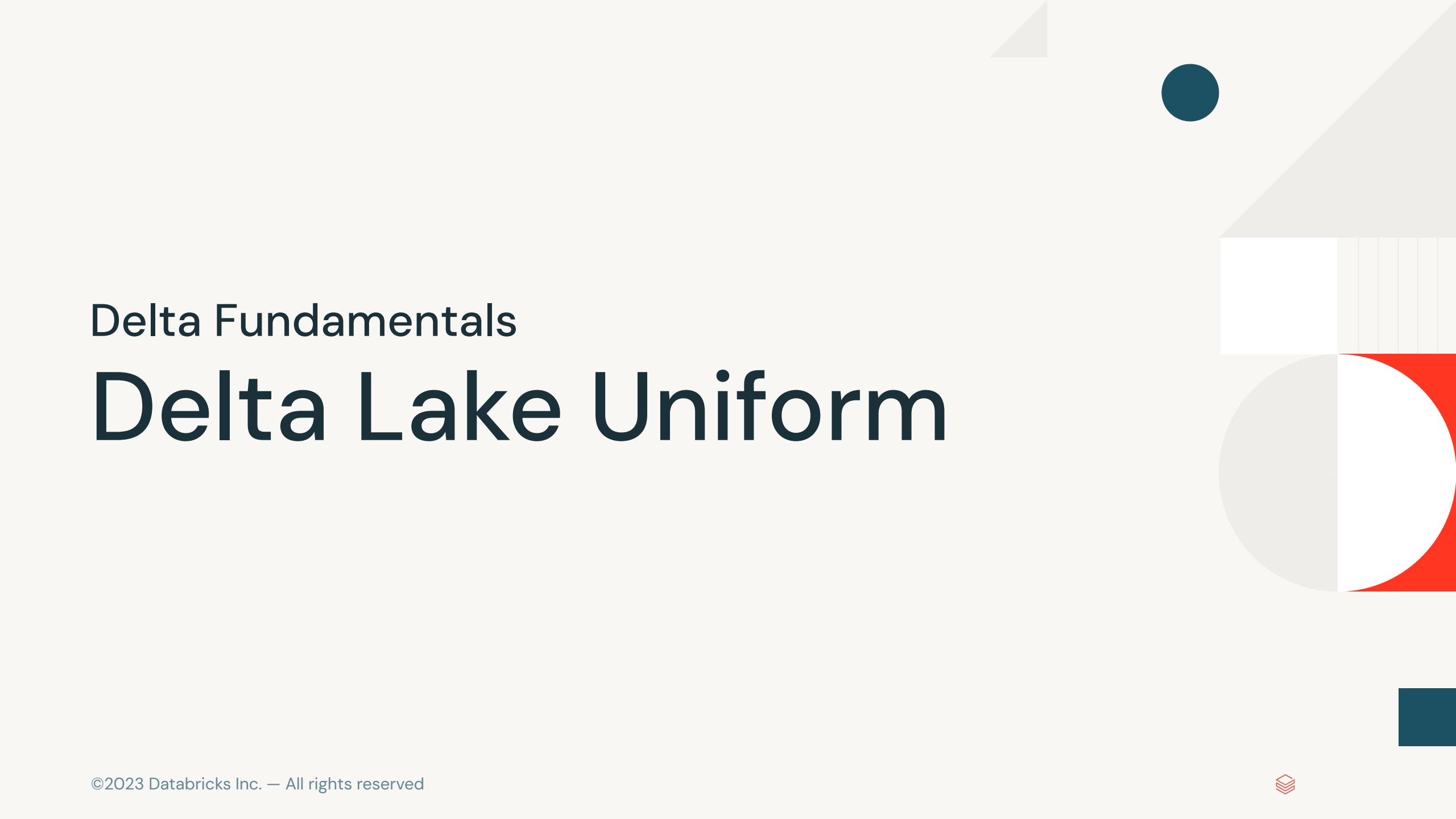
- Deletion Vectors
- Liquid clustering
- Delta Kernel
- Optimized Writes
- Table features
- Incremental checkpoints
- Log compactions
- Row IDs
- Table cloning
- MERGE improvements
- CDF
- Auto-compaction

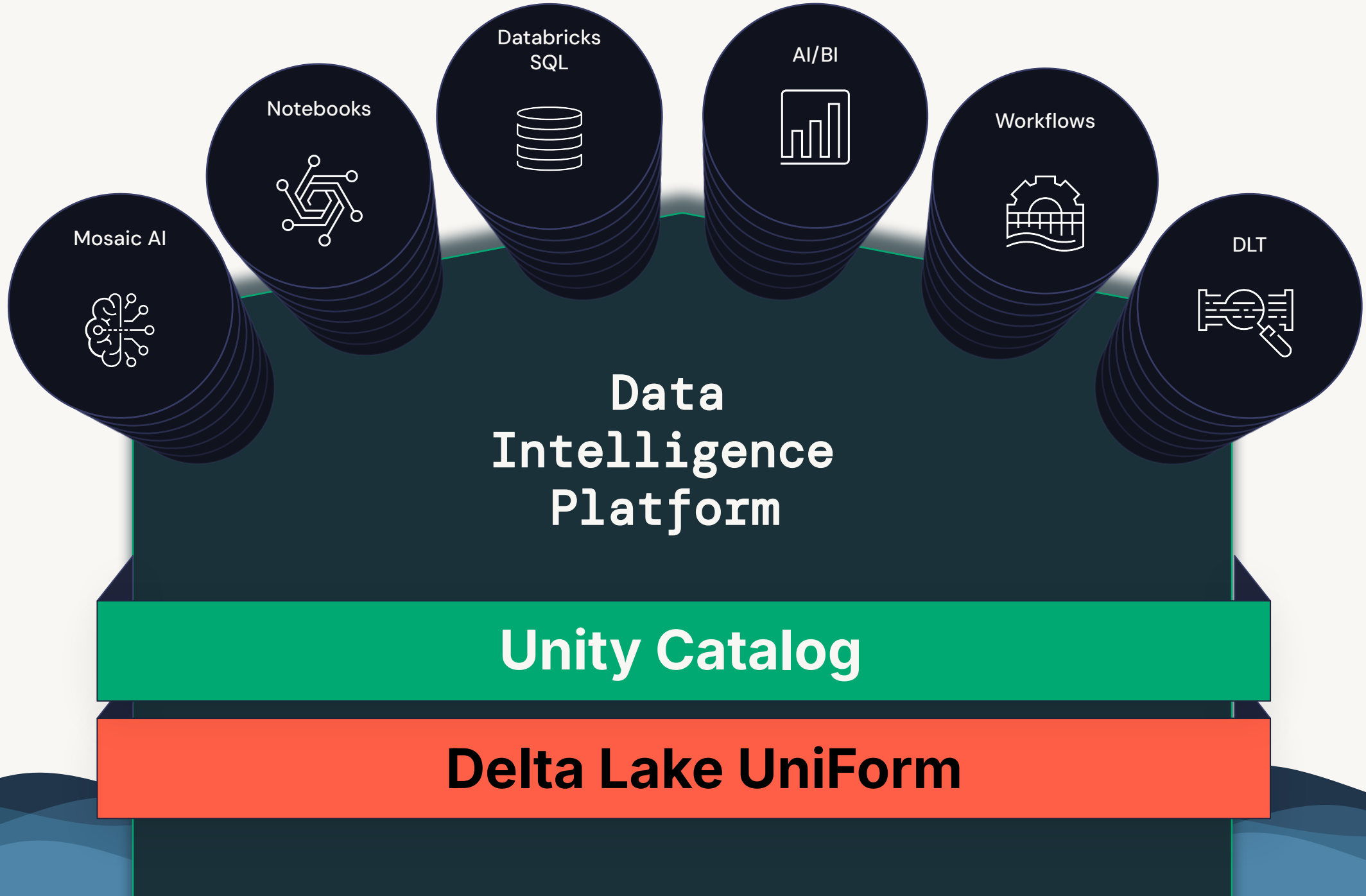
Delta 4.0

- **VARIANT** — Lightning fast semi-structured data
- **UniForm GA** — Write once, read as all formats
- **Liquid GA** — Easy migration from partitioned tables
- **Coordinated Commits** — Cross cloud, cross engines
- **Spark Connect** — Better stability, upgradability
- **Collations** — Flexible sort and comparison
- **Identity columns** — Pain-free primary and foreign keys
- **Type widening** — Data types expand with your data

Delta Fundamentals

Delta Lake Uniform




Mosaic AI
Notebooks
Databricks SQL
AI/BI
Workflows
DLT
Data Intelligence Platform
Unity Catalog
Delta Lake UniForm

Delta Lake UniForm

Dados armazenados em formato Delta podem ser lidos como se fossem Iceberg ou Hudi

Apache Hudi

Metadata

Parquet

Delta Lake

Metadata

Parquet

Apache Iceberg

Metadata

Parquet

Delta Lake UniForm

DELTA LAKE

ICEBERG

databricks

+

Tabular

Evolução de schemas Type Widening & Variant Type

Grande desafio de ETL

Acompanhar o ritmo de mudanças de schema de dados semi-estruturados

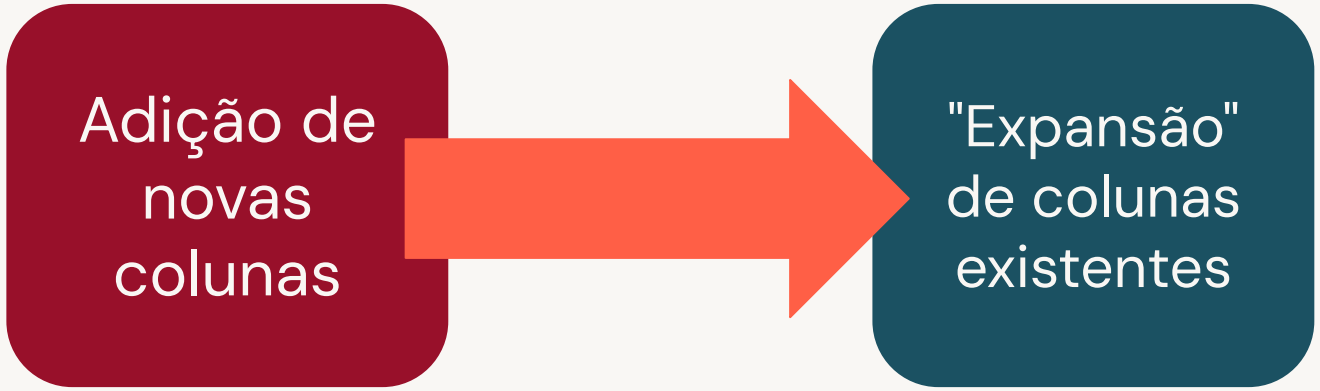
Evolução de schemas

Adição de novas colunas

Evolução de schema automática

- Suportada no INSERT e no MERGE
- Funciona em tipos complexos
- Tipo (type) é imutável quando uma coluna é adicionada

Evolução de schemas



Desafio: O tipo da coluna passa a não comportar os valores

e.g. orderIDs crescem além dos 32-bit do tipo INTEGER escolhido inicialmente

Solução atual: Re-escrever a tabela com novo tipo (LONG)

Problema: Caro e afeta processos em produção

Type Widening!

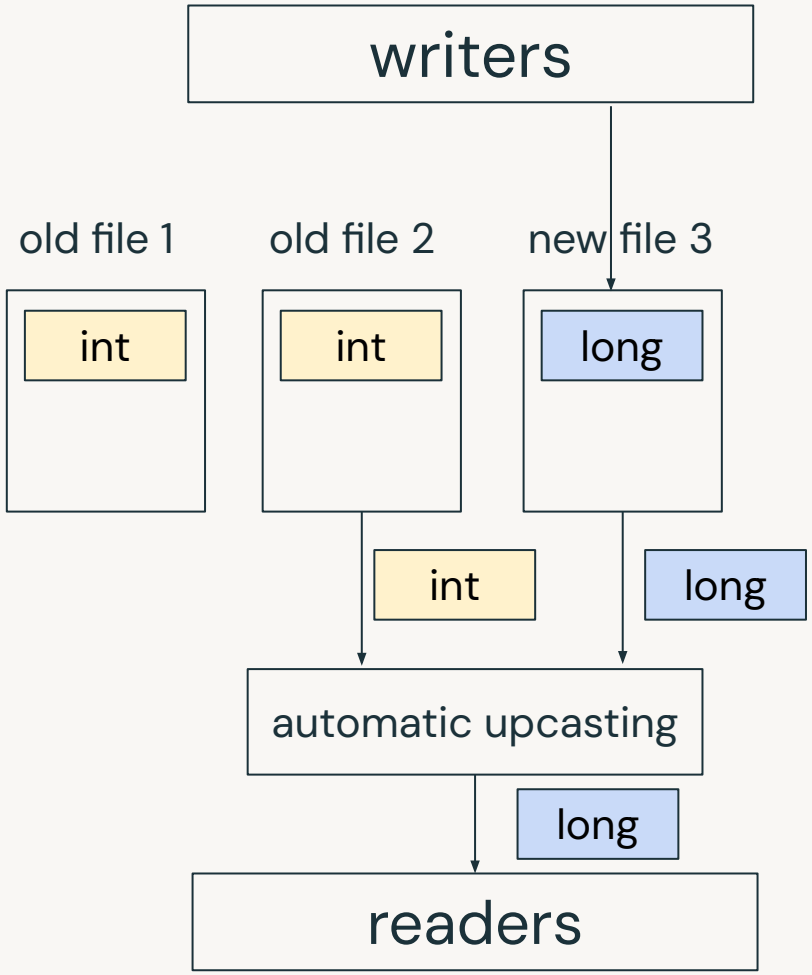
Type Widening

Ideia chave: reconciliação do tipo na leitura

Se **orderId** é modificado de **int** para **long**:

- writers vão escrever novos arquivos parquets com **orderId** sendo **long**
- readers automaticamente fazem o cast dos arquivos antigos como **long**

Não há mais necessidade de **reescrever** os dados existentes!



Evolução da evolução de schemas

Desafio: nova ingestão é feita com tipo completamente diferente

e.g. novo fornecedor entrega orderIDs como texto

Solução atual: escrever os dados JSON em formato text e fazer o parsing na leitura -> Alta flexibilidade, mas leituras muito lentas.

Nova Solução: **Tipo Variant!**

Json exemplo

Como funcionava até então

```
{
  "name": "Chris",
  "age": 23,
  "address": {
    "city": "New York",
    "country": "America"
  },
  "friends": [
    {      "name": "Emily",
      "hobbies": [ "biking", "music", "gaming" ]
    },
    {
      "name": "John",
      "hobbies": [ "soccer", "gaming" ]
    }
  ]
}
```

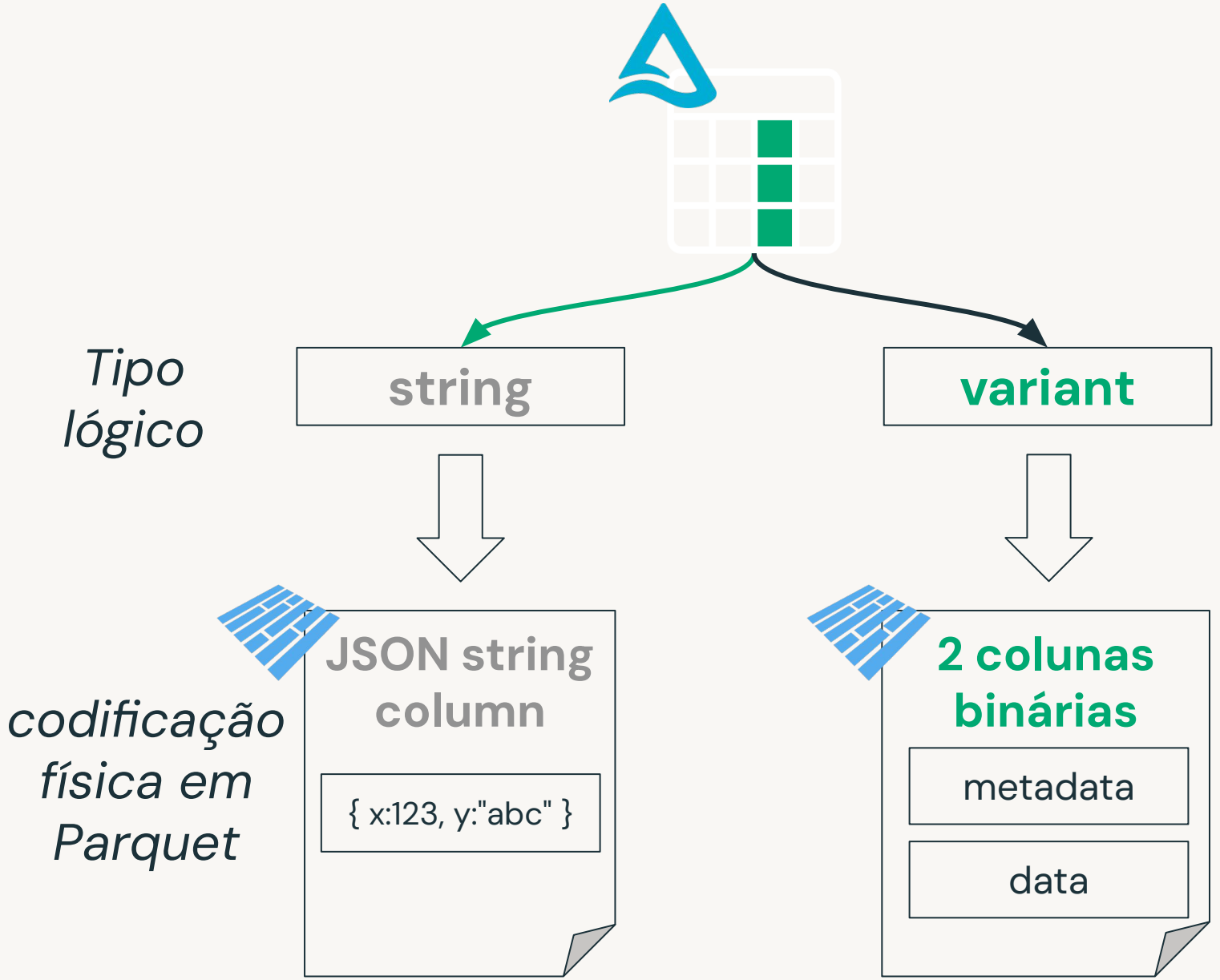
```
StructType([
  StructField("name", StringType(), True),
  StructField("age", IntegerType(), True),
  StructField("address",
    MapType(StringType(),StringType()), True),
  StructField("friends",
    ArrayType(
     MapType(StringType(),
       ArrayType(StringType(),true),true)))

])
```

Variant Type

Novo tipo de dado para armazenar dados semi-estruturados em uma coluna.

Ideia principal: Em vez de salvar o json como string, armazenar com código binário para facilitar a busca de informações.



Variant Type é fácil de utilizar

Funções em Spark para fazer o parsing dos dados

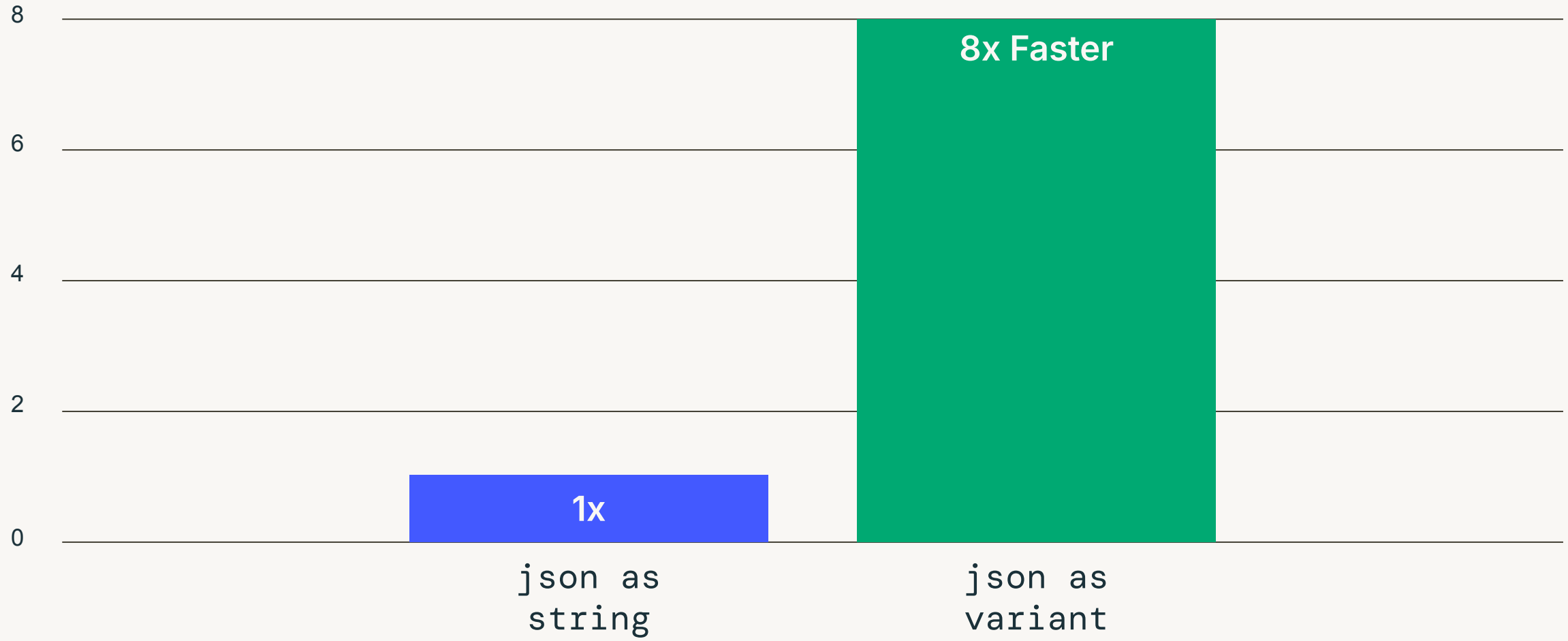
```
INSERT INTO variant_tbl (event_data)
VALUES (PARSE_JSON('{"level": "warning",
"message": "invalid request",
"user_agent": "Mozilla/5.0 ..."}'));
```

Sintaxe de Spark SQL para extrair campos da coluna variante

```
SELECT * FROM variant_tbl
WHERE variant_get(event_data, '$.user_agent', 'string')
like '%Mozilla%'
```

Variant Type é rápido

Melhoria de performance com uso de variante



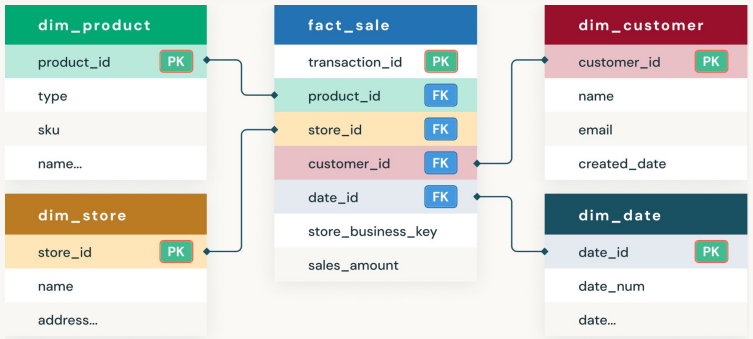
Delta Fundamentals

Identity Columns

Suporte a Identity Columns e Primary/Foreign Keys

Identity Columns

- Geração automática de valores inteiros quando novas linhas são adicionadas à tabela com colunas do tipo "IDENTITY"
- Os usuários também podem inserir explicitamente valores para as colunas IDENTITY



PRIMARY + FOREIGN KEY DECLARATIONS

- Suporte a Chaves Primárias/Estrangeiras (Camada Semântica) com **ALTER TABLE**
- Consulte em "**INFORMATION_SCHEMA**" e "DESCRIBE TABLE"
- Permite o entendimento do relacionamentos entre tabelas

Benefícios: Habilitar qualidade dos dados e descoberta de relacionamentos entre tabelas por ferramentas e usuários que não estão familiarizados com o modelo de dados.

Delta Fundamentals

MERGE

De que característica estamos falando?

```
MERGE INTO customers     -- Delta table
USING source             -- Incremental dataset
ON customers.customerId = source.customerId
WHEN MATCHED AND customers.customerId < 100 THEN
    DELETE *
WHEN MATCHED THEN
    UPDATE SET address = source.address
WHEN NOT MATCHED
    THEN INSERT (customerId, address) VALUES (source.customerId,
source.address)
```

Definição

```
MERGE INTO target_table_identifier [AS target_alias]
USING source_table_identifier [<time_travel_version>] [AS source_alias]
ON <merge_condition>
[ WHEN MATCHED [ AND <condition> ] THEN <matched_action> ]
[ WHEN MATCHED [ AND <condition> ] THEN <matched_action> ]
[ WHEN NOT MATCHED [ AND <condition> ]  THEN <not_matched_action> ]
```

```
MERGE INTO events
USING updates
ON events.eventId = updates.eventId
WHEN MATCHED THEN
  UPDATE SET events.data = updates.data
WHEN NOT MATCHED
  THEN INSERT (date, eventId, data) VALUES (date, eventId, data)
```

```
DeltaTable.forPath(spark, "/data/events/")
  .as("events")
  .merge(
    updatesDF.as("updates"),
    "events.eventId = updates.eventId")
  .whenMatched
  .updateExpr(
    Map("data" -> "updates.data"))
  .whenNotMatched
  .insertExpr(
    Map(
      "date" -> "updates.date",
      "eventId" -> "updates.eventId",
      "data" -> "updates.data"))
  .execute()
```

Etapas de um MERGE (grosso modo)

tabela de atualização e tabela de destino

1. Junção interna entre atualização e destino usando a condição na cláusula ON → Retorna ao driver: lista de arquivos no **destino** que contêm linhas correspondentes
2. O curto-circuito acontece:
   - Se NÃO houver linhas correspondentes no **destino** na etapa 1, use uma gravação somente acréscimo para acrescentar **atualização** ao **destino**
   - Else: Use um full-outer-join para consolidar todas as alterações entre a **atualização** e o **destino**
3. Confirme atomicamente no log de transações Delta, removendo os arquivos correspondentes da etapa 1 e adicionando

Para um entendimento mais profundo:
https://youtu.be/7ewmcdrylsA

O que acontece se você fizer isso repetidamente (por exemplo, em streaming estruturado usando foreachBatch)?

Efeitos colaterais de um MERGE

- Se não tivermos uma condição específica o suficiente para MATCH, estamos reescrevendo muitos dados!

- Se tivermos uma condição específica o suficiente, criamos muitos arquivos pequenos todas as vezes (porque as junções usam `spark.sql.shuffle.partitions = 200` por padrão!) → isso eventualmente interromperá a etapa 1 do processo MERGE quando tiver que enviar milhões de caminhos de arquivo para o driver

O que acontece se você fizer isso repetidamente (por exemplo, em streaming estruturado usando foreachBatch)?

Efeitos colaterais de um MERGE

- **Soluções**:
  - Temos que pensar sobre o layout de dados da tabela de destino para que não correspondamos a 90% + dos arquivos todas as vezes
  - Podemos ajustar a configuração `spark.sql.shuffle.partitions` antes de executar o optimize (ao custo do paralelismo de gravação), além de executar `OPTIMIZE` com mais frequência.
    - Um recurso chamado "Otimização automática" (TBD mais tarde na sessão) lida com isso automaticamente.

O que acontece se você fizer isso repetidamente (por exemplo, em streaming estruturado usando foreachBatch)?

Workflows

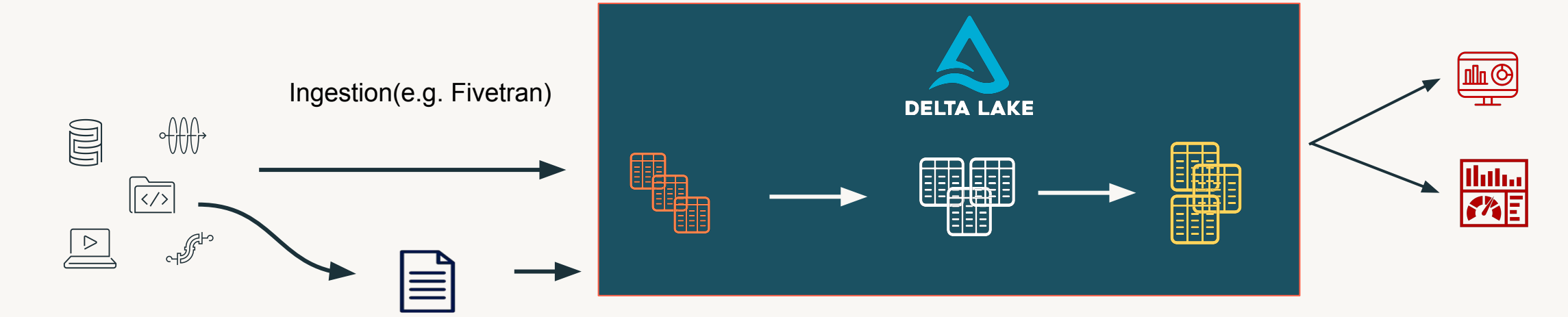
Batch Processing Overview

Batch Processing

- Generally refers to processing & analysis of ***bounded*** datasets
  - Size is well known
  - We can count the number of elements
- Typical of applications where there are loose data latency requirements
  - (ie. day old, week old, month old).
- This was traditional ETL from transactional systems into analytical systems.

Batch Processing

Traditional data processing



Streaming Processing

Why is stream processing getting popular?

Data Velocity/Volumes

Rising data velocity & volumes require continuous, incremental processing - **impossible to process all data in batch in time.**

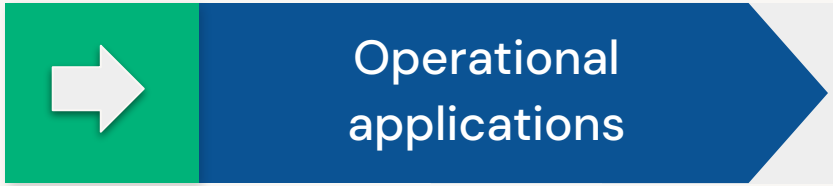
Real-time analytics

Business demands access to up-to-date data for faster, better business decisions. **Fresh data is more actionable**

Operational applications

Critical operational applications need real-time data for effective, **instantaneous response**

Why is stream processing getting popular?



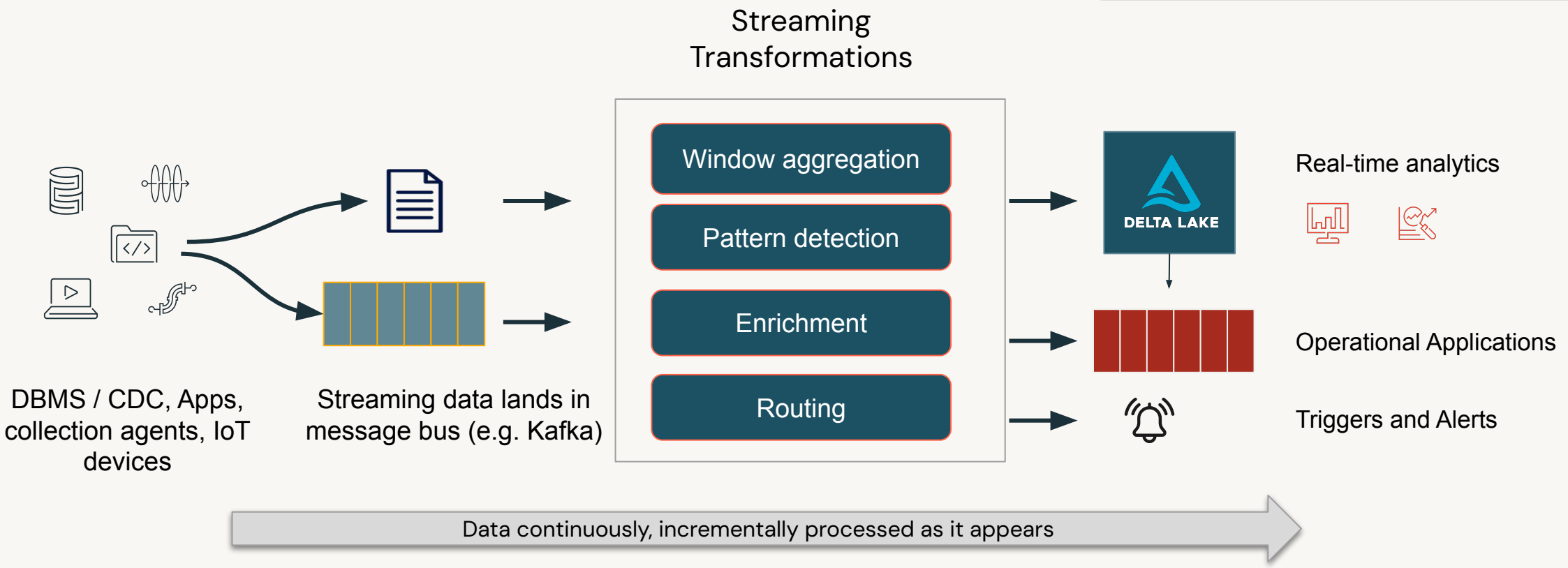
Modern Use Cases

- Fraud detection
- Processing IoT sensor data
- Real-time dashboards
- Network / equipment monitoring
- Digital advertising
- Geofencing, vehicle tracking
- Cybersecurity

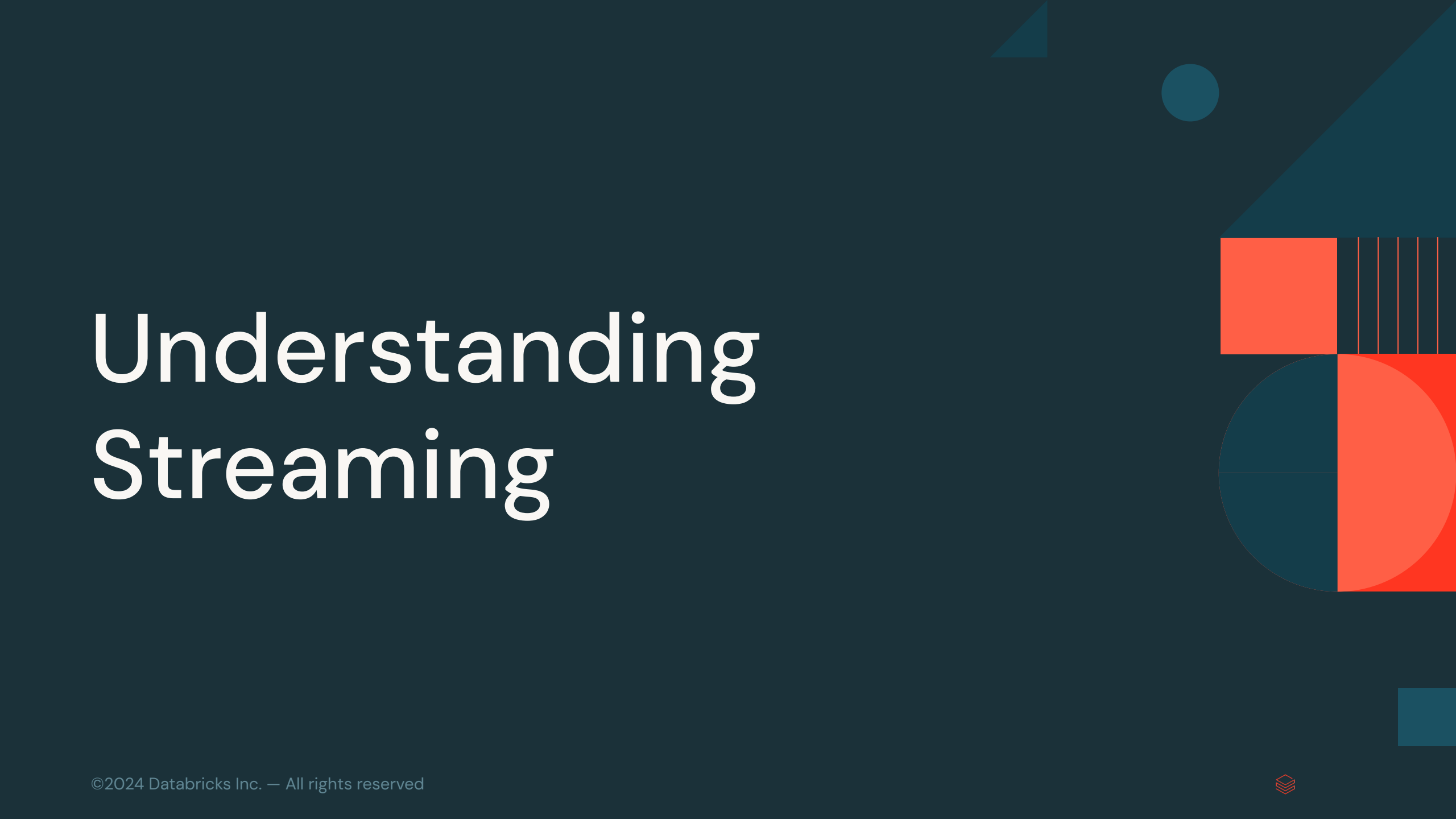
Stream Processing

Modern data processing

Most data is created as a series of events over time: e.g. transactions, sensor events, user activity on a website



Understanding Streaming

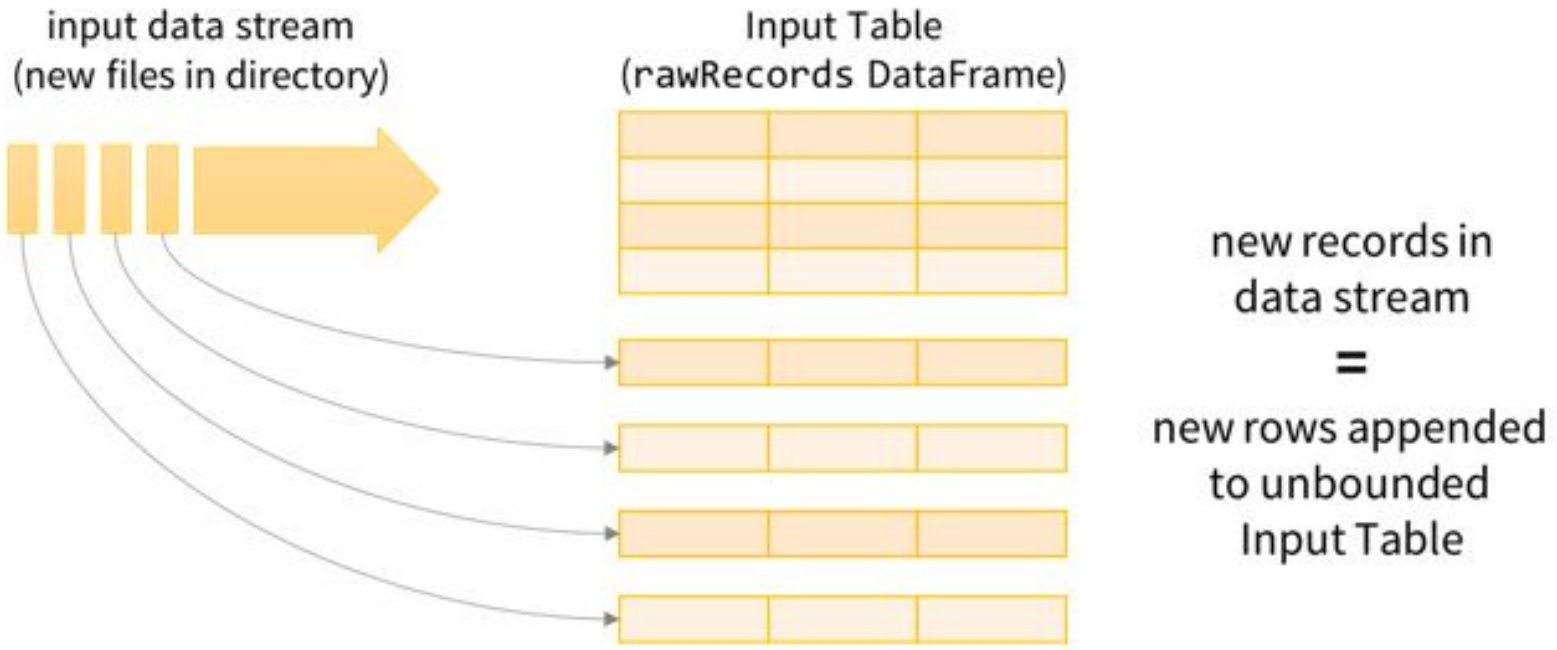


What is streaming?

- When people talk about streaming they typically refer to:
  - a processing engine that can process...
  - ...incoming streams of data

- There are actually 2 orthogonal concepts to distinguish:
  - stream/batch processing engines, and
  - bounded/unbounded datasets

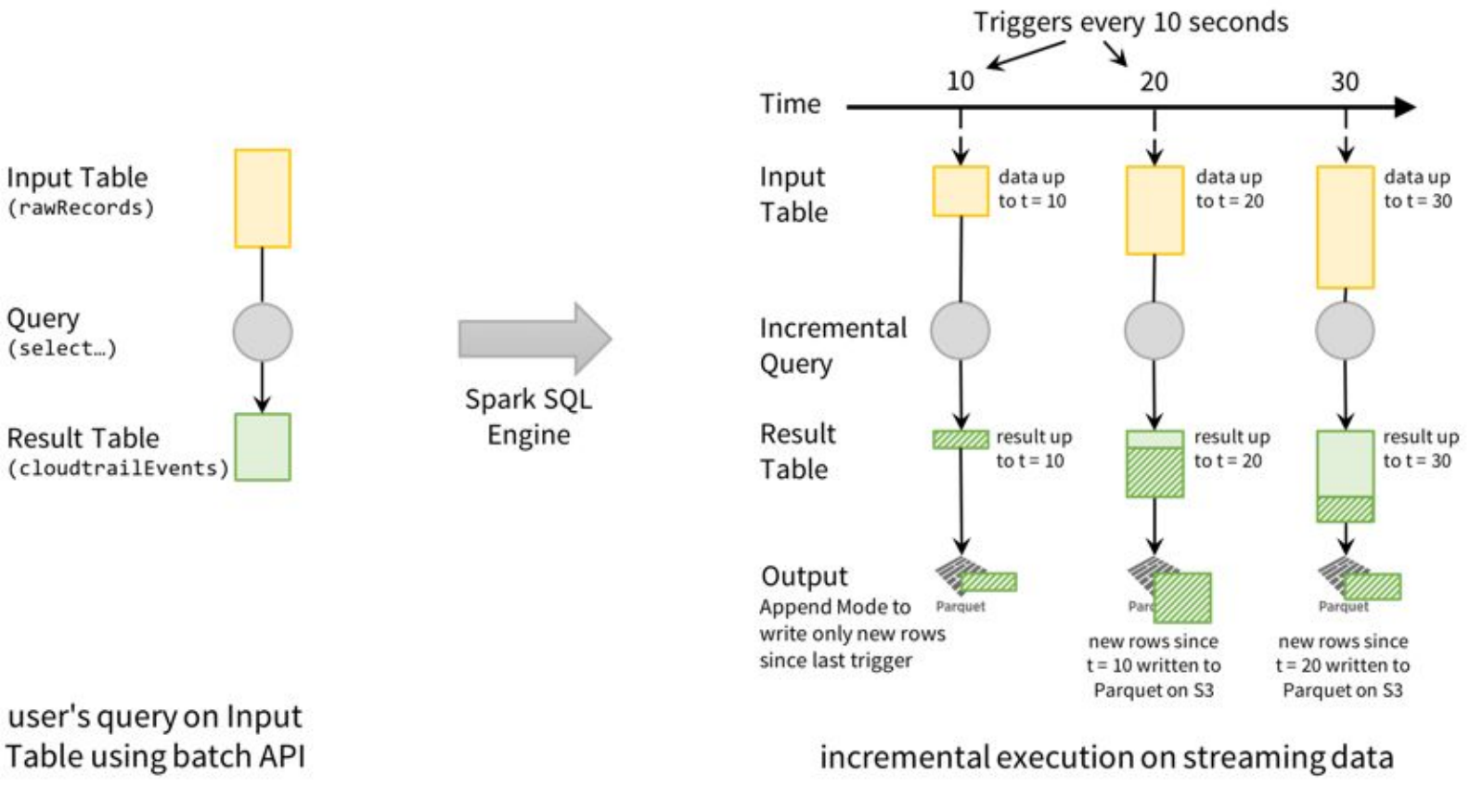
Streams a unbounded tables



Structured Streaming Model

treat data streams as unbounded tables

Structured Streaming – Execution



Structured Streaming Model

users express query on streaming data using a batch API; Spark incrementalizes them to run on streams

Analogy: Taking care of your baggage

Objective: Making sure you do have everything you packed when you arrive at your destination

Method: Counting your clothes before and after

When at home you count your clothes

Human - batch processing engine

Stuff - bounded dataset

You consider all bags you take with you

Still a bounded dataset, can consider suitcases as mini-batches

At the Baggage belt counting clothes

Unbounded dataset, can consider suitcases as mini-batches and baggage belt as stream processing engine

Why use streaming (vs. batch) ?

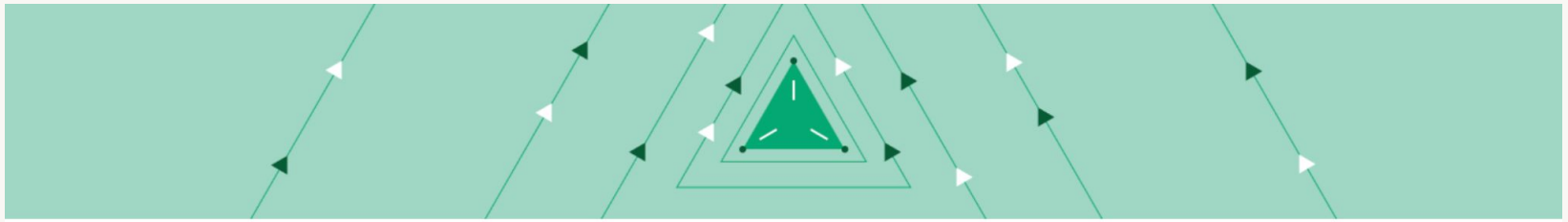
Streaming can handle a superset of the use cases:

- it can deal with incomplete (unbounded) data
- it can provide lower latency
- it automatically does bookkeeping for you
  - (what input still needs to be processed)

But won't that be very expensive, requiring always on clusters?

Not necessarily: Databricks allows you to schedule a stream to only process the current available data

Run streaming as scheduled jobs: 10x savings



Running Streaming Jobs Once a Day For 10x Cost Savings

Part 6 of Scalable Data @ Databricks

by Burak Yavuz and Tyson Condie

May 22, 2017 in Engineering Blog

Share this post

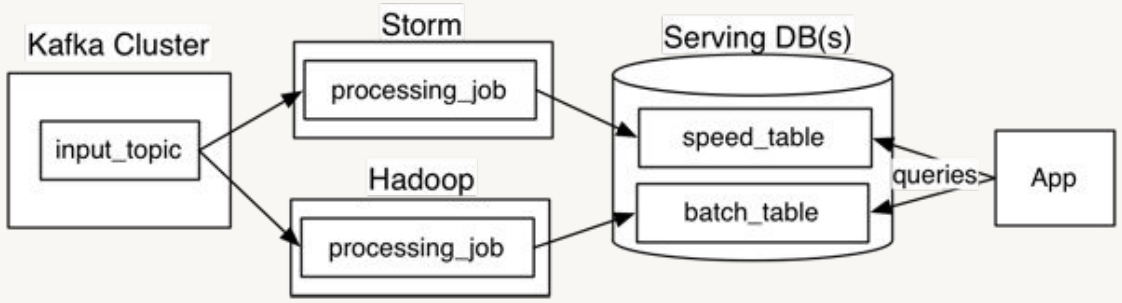
This is the sixth post in a multi-part series about how you can perform complex streaming analytics using Apache Spark.

Traditionally, when people think about streaming, terms such as “real-time,” “24/7,” or “always on” come to mind. You may have cases where data only arrives at fixed intervals. That is, data appears every hour or once a day. For these use cases, it is still beneficial to

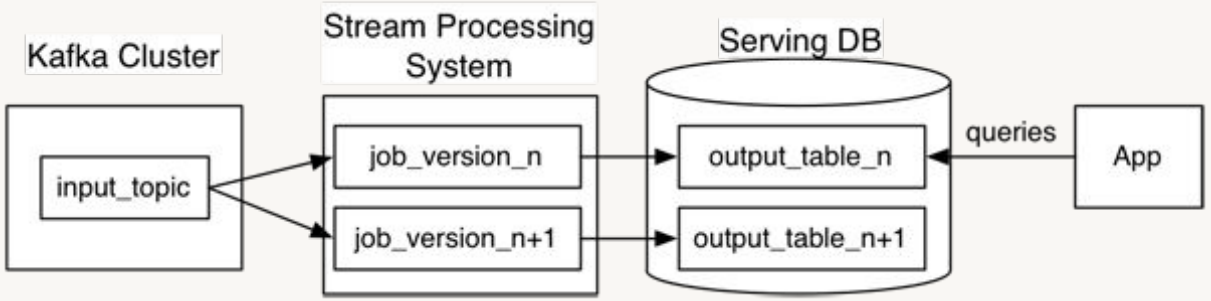
Data Lake Challenges for Streaming Data

Lambda & Kappa

- Lambda Architecture
  - Nathan Marz (Creator of Apache Storm) in 2011 - How to beat the CAP theorem



- Kappa Architecture
  - Jay Kreps (Confluent Co-founder) - Questioning the Lambda Architecture

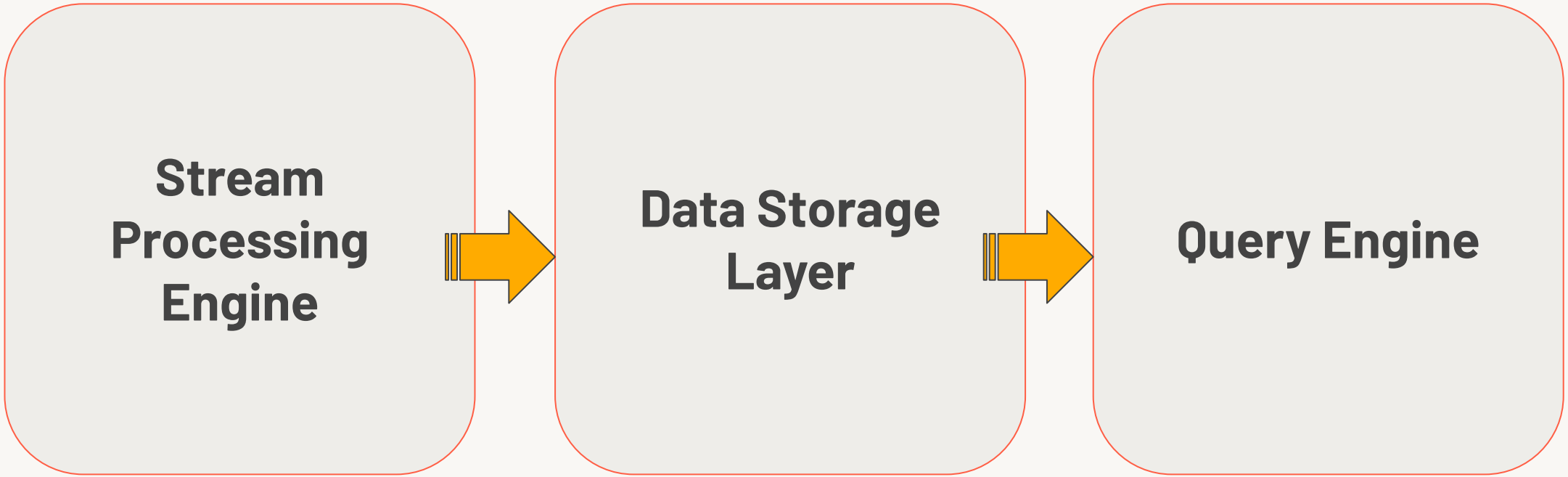


Lambda & Kappa: Complexity

- **Lambda**
  - **Write your code twice** – in two different complex, distributed frameworks
  - Operationalize, maintain, and debug **two different complex, distributed systems**
  - You are essentially "*abstracting over totally divergent programming paradigms built on top of barely stable distributed systems*"
    - Jay Kreps, 2014 (Co-founder, Confluent)
- **Kappa**
  - **Increased retention of your source event logs** (ie, Kafka retention to 30+ days).
  - You have to **reprocess data to recompute results.**
  - You have to "cut-over" your target tables.

Simplifying the Lambda & Kappa Architectures

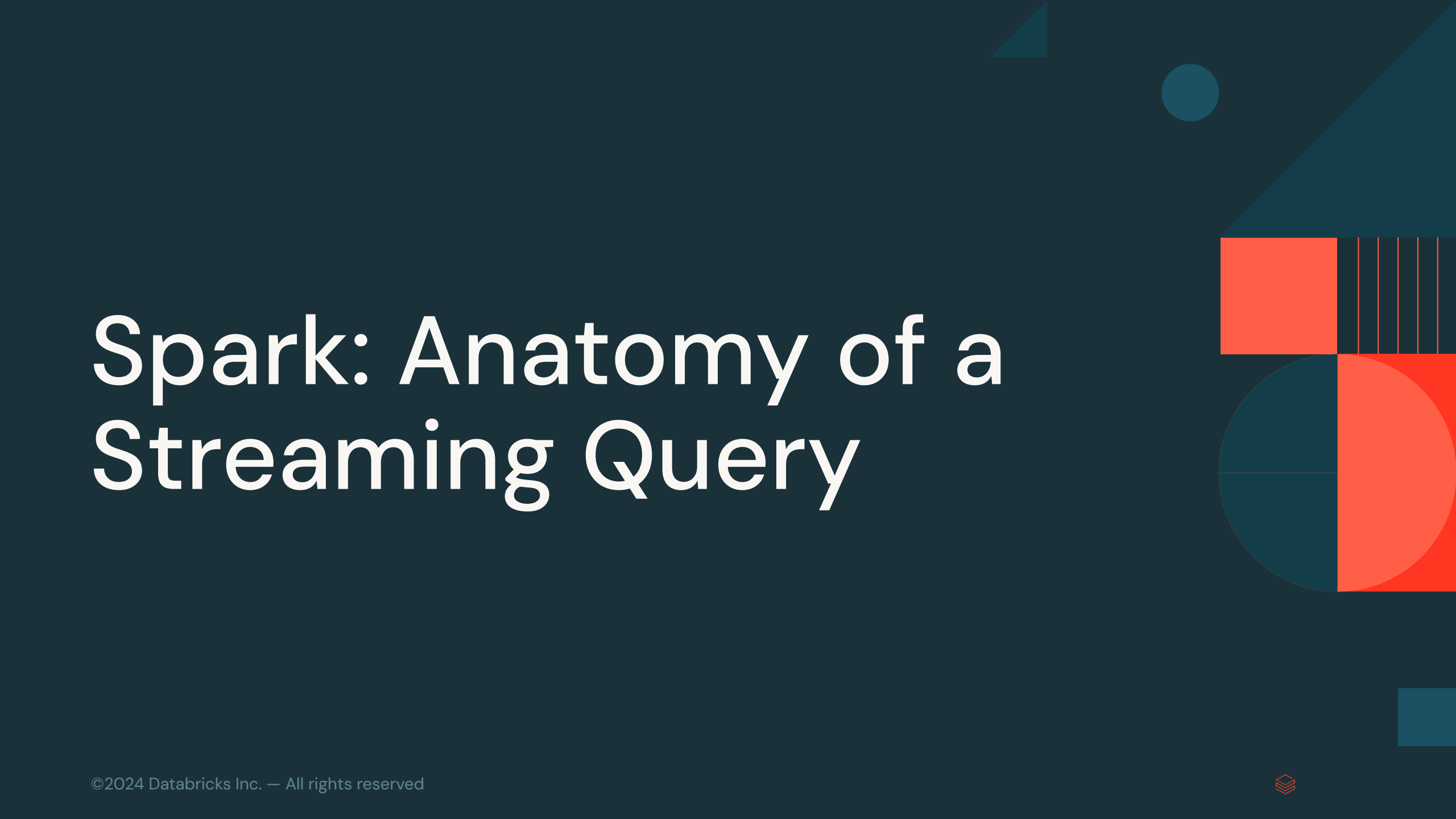
- *3 components for a simplified, minimum architecture*



Introducing the Delta Architecture

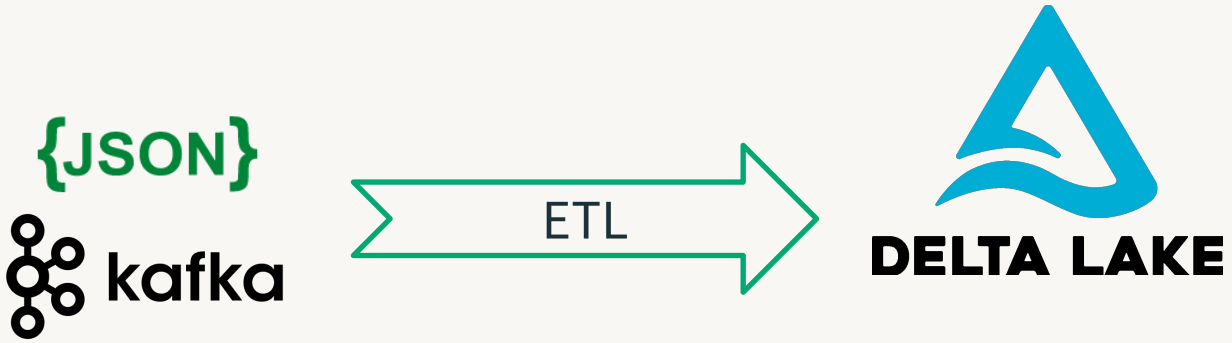
- Scalability + Reliability + Performance

Spark: Anatomy of a Streaming Query



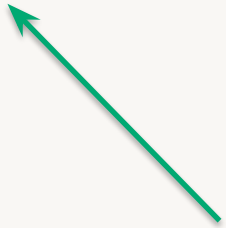
Spark: Anatomy of a Streaming Query

- Example:
  - Read JSON data from Kafka
  - Parse nested JSON
  - Store in structured Delta Lake table
  - Get end-to-end failure guarantees



Anatomy of a Streaming Query

```
spark.readStream.format("kafka")
  .option("kafka.bootstrap.servers",...)
  .option("subscribe", "topic")
  .load()
```



returns a Spark DataFrame
(common API for batch & streaming data)

Source:

- Specify where to read data from
- Built-in support for Files / Kafka / Kinesis / EventHubs
- Can include multiple sources of different types using join() / union()

Anatomy of a Streaming Query

```
spark.readStream.format("kafka")
  .option("kafka.bootstrap.servers",...)
  .option("subscribe", "topic")
  .load()
```

Kafka

key	value	topic	partition	offset	timestamp
[binary]	*[binary]*	"topic"	0	345	1486087873
[binary]	*[binary]*	"topic"	3	2890	1486086721

Anatomy of a Streaming Query

```
spark.readStream.format("kafka")
  .option("kafka.bootstrap.servers",...)
  .option("subscribe", "topic")
  .load()
  .selectExpr("cast (value as string) as json")
```

Transformations

- Cast bytes from Kafka records to a string, parse it as a json, and generate nested columns
- 100s of built-in, optimized SQL functions like from_json
- user-defined functions, lambdas, function literals with map, flatMap...

Anatomy of a Streaming Query

```
spark.readStream.format("kafka")
  .option("kafka.bootstrap.servers",...)
  .option("subscribe", "topic")
  .load()
  .selectExpr("cast (value as string) as json")
  .writeStream
  .format("delta")
  .option("path", "/deltaTable/")
```

Sink

- Write transformed output to external storage systems
- Built-in support for Files / Kafka
- Use foreach to execute arbitrary code with the output data
- Some sinks are transactional and exactly once (e.g. files)

Anatomy of a Streaming Query

```
spark.readStream.format("kafka")
  .option("kafka.bootstrap.servers",...)
  .option("subscribe", "topic")
  .load()
  .selectExpr("cast (value as string) as json")
  .writeStream
  .format("delta")
  .option("path", "/deltaTable/")
  .trigger("1 minute")
  .option("checkpointLocation", "...")
  .start()
```

Processing Details

- Trigger: when to process data
  - Fixed interval micro-batches
  - As fast as possible micro-batches
- Checkpoint location: for tracking the progress of the query

Bonus: Segregate Flow/Transforms

```
spark.readStream.format("kafka")
  .option("kafka.bootstrap.servers",...)
  .option("subscribe", "topic")
  .load()
  .transform(castPayloadToJSON)
  .writeStream
  .format("delta")
  .option("path", "/deltaTable/")
  .trigger("1 minute")
  .option("checkpointLocation", "...")
  .start()

def castPayloadToJSON(df: DataFrame): DataFrame = {
  df.selectExpr("cast (value as string) as json")
}
```

Use Transform()

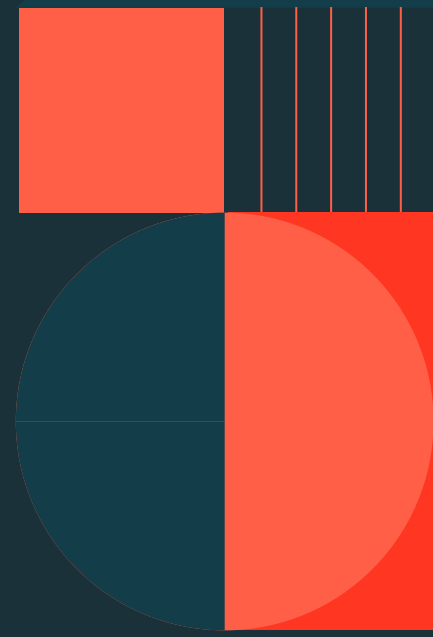
- Simple function
  - Receives a DataFrame
  - Returns a DataFrame
- Segregates Streaming Flow from Processing Logic
- Much easier to construct Unit Tests!
  - Example here.

Demo

05 - Streaming e Variant

Data Operations on Streaming Data

Simplifying Data Ingestion with Auto Loader



Benefits of Auto Loader

- No Custom Bookkeeping: Incrementally process new files as they land in object storage. Customers don't need to manage any state information on what files arrived.
- Scalable: Efficiently track the new files arriving by leveraging cloud notification and queue services without having to list all the files in a directory. Scalable to handle millions of files in a directory.
- Easy to use: Automatically sets up cloud notification and queue services required.

Streaming Loads with Auto Loader

```
CREATE STREAMING TABLE raw_data
AS SELECT *
FROM cloud_files ("/raw_data", "json")
```

```
spark.readStream
     .format("cloudFiles")
     .option("cloudFiles.format", "json")
     .schema(schema)
     .load("/input/path")
     .writeStream
     .format("delta")
     .option("checkpointLocation", "/checkpoint/path")
     .start("/out/path")
```

Batch Loads with Auto Loader

```
spark.readStream
     .format("cloudFiles")
     .option("cloudFiles.format", "json")
     .schema(schema)
     .load("/input/path")
     .writeStream
     .format("delta")
     .trigger(Trigger.Once) // or Trigger.AvailableNow
     .option("checkpointLocation", "/checkpoint/path")
     .start("/out/path")
```

New File Detection Modes

Directory Listing Mode

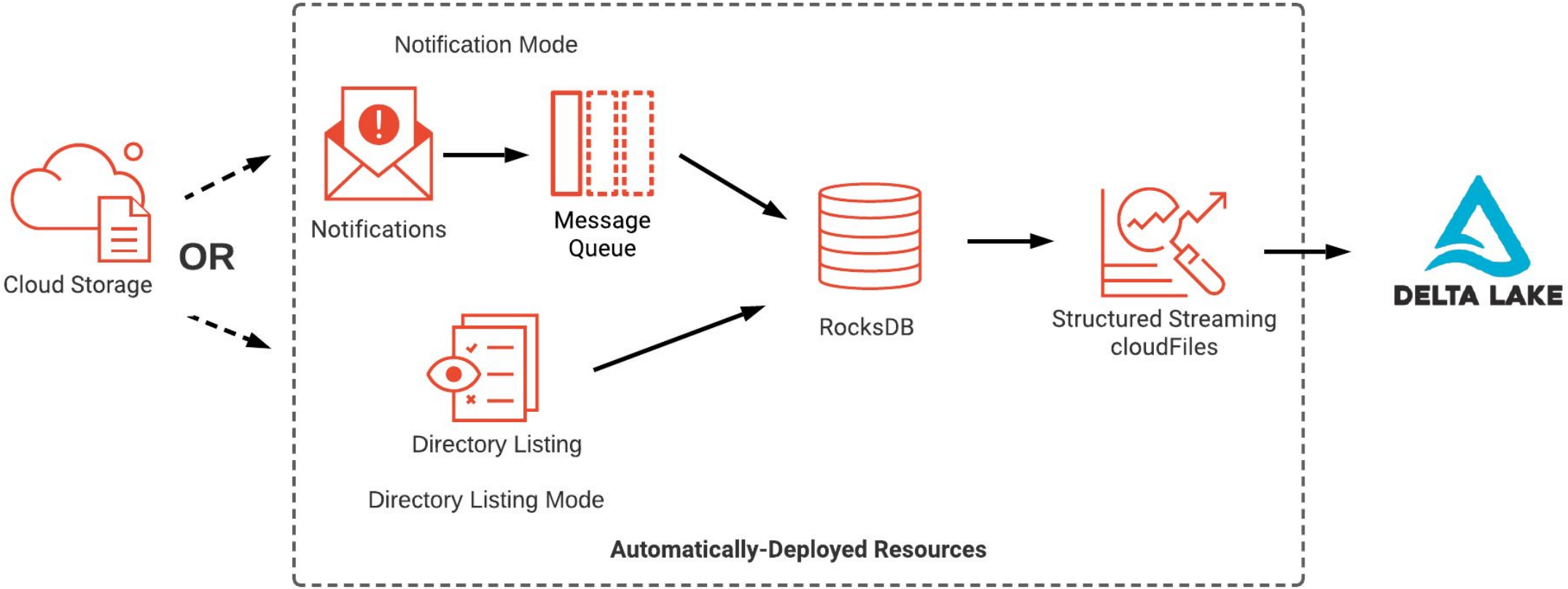
- Default mode
- Easily stream files from object storage without configuration
- Creates file queue through parallel listing of input directory
- Good for smaller source directories

File Notification Mode

- Requires some security permissions to other cloud services
- Uses cloud storage queue service and event notifications to track files
- Configurations handled automatically by Databricks
- Scales well as data grows

Auto Loader Under the Hood

```
.option("cloudFiles.useNotifications","true")
```



Reasoning about Time

Event Time vs. Processing Time

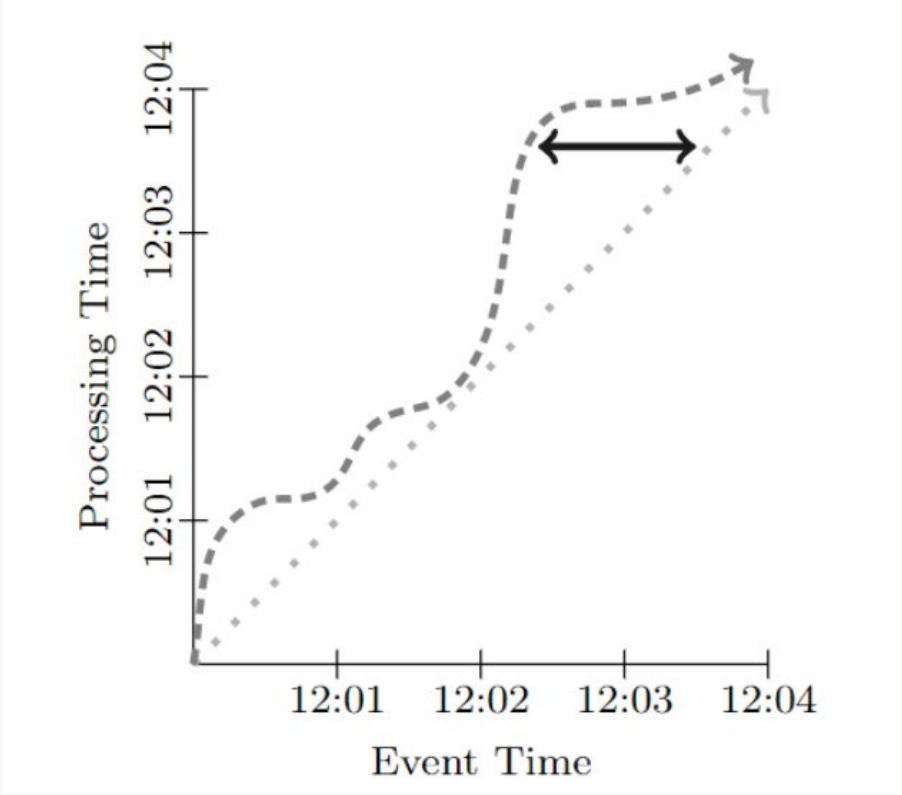
Event Time

Time at which the event (record in the data) actually occurred.

Processing time

Time at which a record is actually processed.

Time Domain Skew

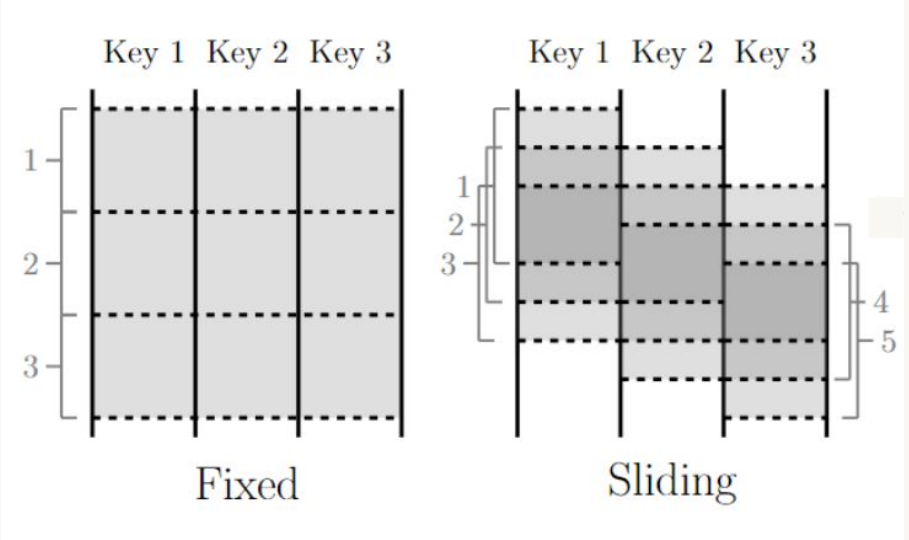


Types of Stream Processing

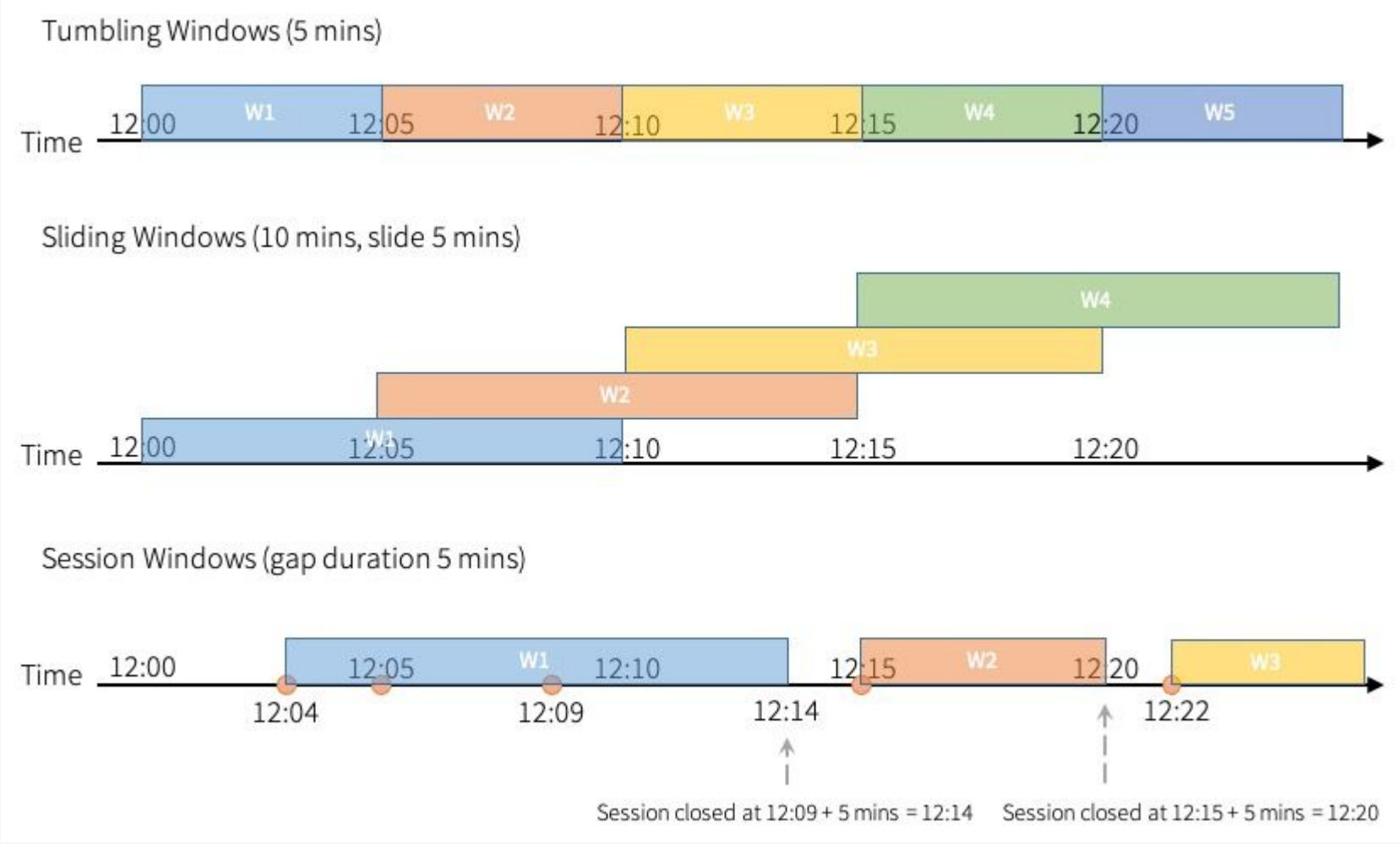
- **Stateless**
  - Typically trivial transformations. The way records are handled do not depend on previously seen records.
  - Example: Data Ingest (map-only), simple dimensional joins
- **Stateful**
  - **Previously seen records can influence new records**
  - Example: Aggregations over time, Fraud/Anomaly Detection

Windows for Streaming Aggregation

- Provide easy syntax to get near real-time metrics on data as it's processed
- Fixed windows aggregate all events in a set period of time:
  - How many events per 10 minute block in a given hour
  - 12:00-12:10, 12:10-12:20, 12:20-12:30
- Sliding windows overlap:
  - How many events per 10 minute block starting every 2 minutes
  - 12:00-12:10, 12:02-12:12, 12:04-12:14



Windows for Streaming Aggregation



Late Data and Watermarking

- Watermarks define how long we'll continue to update each window
- Late data arriving within this threshold will be used to update results
- Late data arriving outside of this threshold will not be included in the windowed aggregation

Late Data and Watermarking

```
spark.readStream.format("kafka")
  .option("kafka.bootstrap.servers",...)
  .option("subscribe", "topic")
  .load()
  .transform(castPayloadToJSON)
  .dropDuplicates("id")
  .writeStream
  .format("delta")
  .option("path", "/deltaTable/")
  .trigger("1 minute")
  .option("checkpointLocation", "...")
  .start()
```

This is very dangerous!

- No watermarks
  - State will grow without any limits
- Essentially all lines will be kept in the checkpoint forever!
- This streaming job will crash very soon!

Late Data and Watermarking

```
spark.readStream.format("kafka")
  .option("kafka.bootstrap.servers",...)
  .option("subscribe", "topic")
  .load()
  .transform(castPayloadToJSON)
  .withWatermark("eventTime", "10 minutes")
  .dropDuplicates("id", "eventTime")
  .writeStream
  .format("delta")
  .option("path", "/deltaTable/")
  .trigger("1 minute")
  .option("checkpointLocation", "...")
  .start()
```

Watermark defined

- State will be limited to this window
- However, both columns must be real duplicates
- If data arrives later than 10 minutes, it will be sent forward
- Can be combined with MERGE

Joining Streaming and Static Tables

Joining Two Streams

- **STATEFUL**: Need to specify watermark with time constraints for correct results and state cleanup
- Conditionally supports most join types
  - Inner
  - Left
  - Right
  - Full
  - Left Semi

Stream-Static Joins

- **STATELESS**: No state information is tracked
- Support driven by the streaming table
  - Inner
  - Left Outer
  - Left Semi
- For each microbatch of streaming records, join with static table

Updating Tables with Streaming Data

Streaming Upserts with Delta Lake

Basic Upsert

```
MERGE INTO delta_table a
USING new_data b
ON a.key = b.key
WHEN MATCHED
   THEN UPDATE SET *
WHEN NOT MATCHED
   THEN INSERT *
```

Insert-Only Merge

```
MERGE INTO delta_table a
USING new_data b
ON a.key = b.key
WHEN NOT MATCHED
   THEN INSERT *
```

foreachBatch

- Transforms a streaming microbatch into a batch operation
- Allows use of batch data writers with streaming data
- Allows writing to multiple locations
- Documentation presents examples for SCD Types 1 and 2 here.
- Example here.

Propagating Changes with Delta Change Data Feed

What is Stream Composability?

- Structured Streaming expects append-only sources
- Delta tables are composable if new streams can be initiated from them

Operations that break stream composability

- Complete aggregations
- Delete
- UPDATE/MERGE

Data is changed in place, breaking append-only expectations

Current Challenges

Identifying Changes

Updates in ETL struggle to find changes in the data from version to version in large tables

Without information regarding the specific changes to be made, all data must be compared

Updating BI & Analytics Data

Real-time updates to BI and analytics require additional processing as changes arrive

Recalculating full datasets causes downtime to users incompatible with real-time needs

Producing an Audit Trail

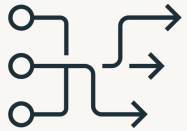
Audits of records, en masse or individually, demand the ability to readily construct data as it was at any or every point in time

Digging through all versions is impractical yet required to meet compliance requirements

What Delta Change Data Feed Does for You

Improve ETL pipelines

Process less data during ETL to increase efficiency of your pipelines



Unify batch and streaming

Common change format for batch and streaming updates, appends, and deletes

BI on your data lake

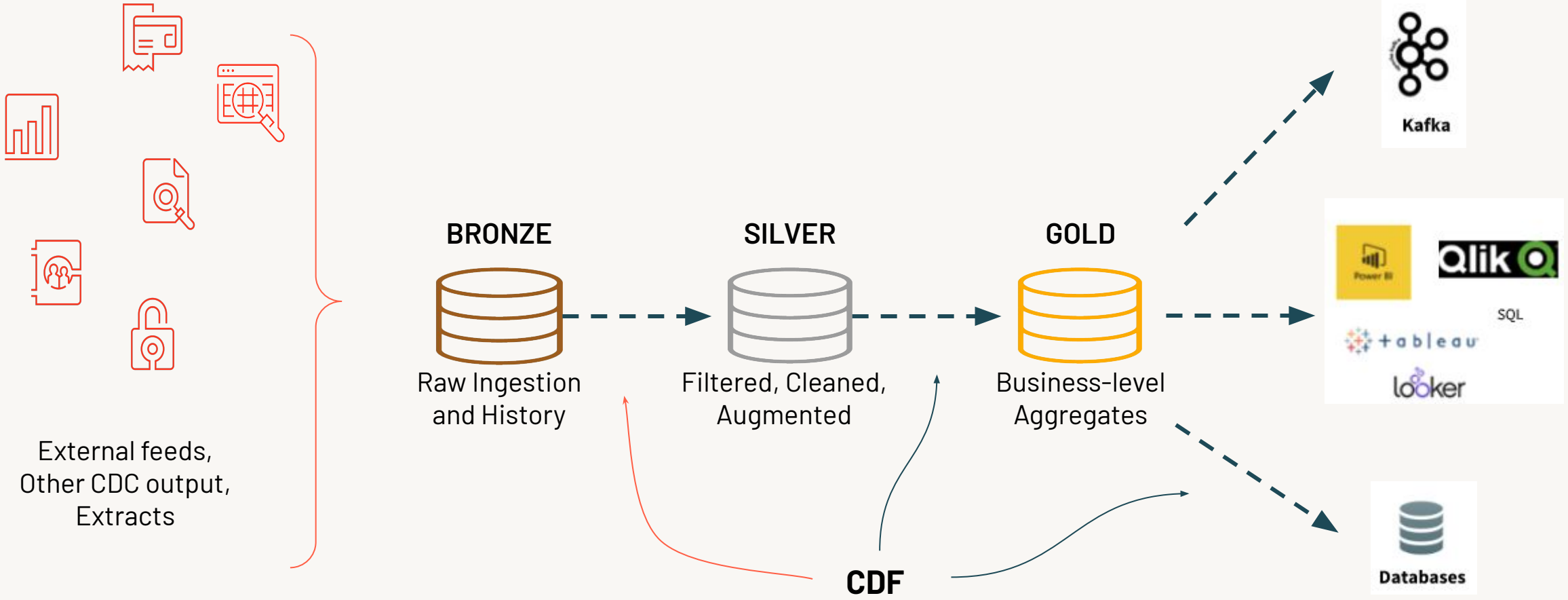
Incrementally update the data supporting your BI tool of choice

Meet regulatory needs

Full history available of changes made to the data, including deleted information

Delta Change Data Feed

Where Delta Change Data Feed Applies



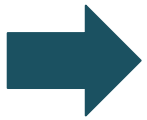
How Does Delta Change Data Feed Work?

Original Table (v1)

PK	B
A1	B1
A2	B2
A3	B3

Change data (Merged as v2)

PK	B
A2	Z2
A3	B3
A4	B4

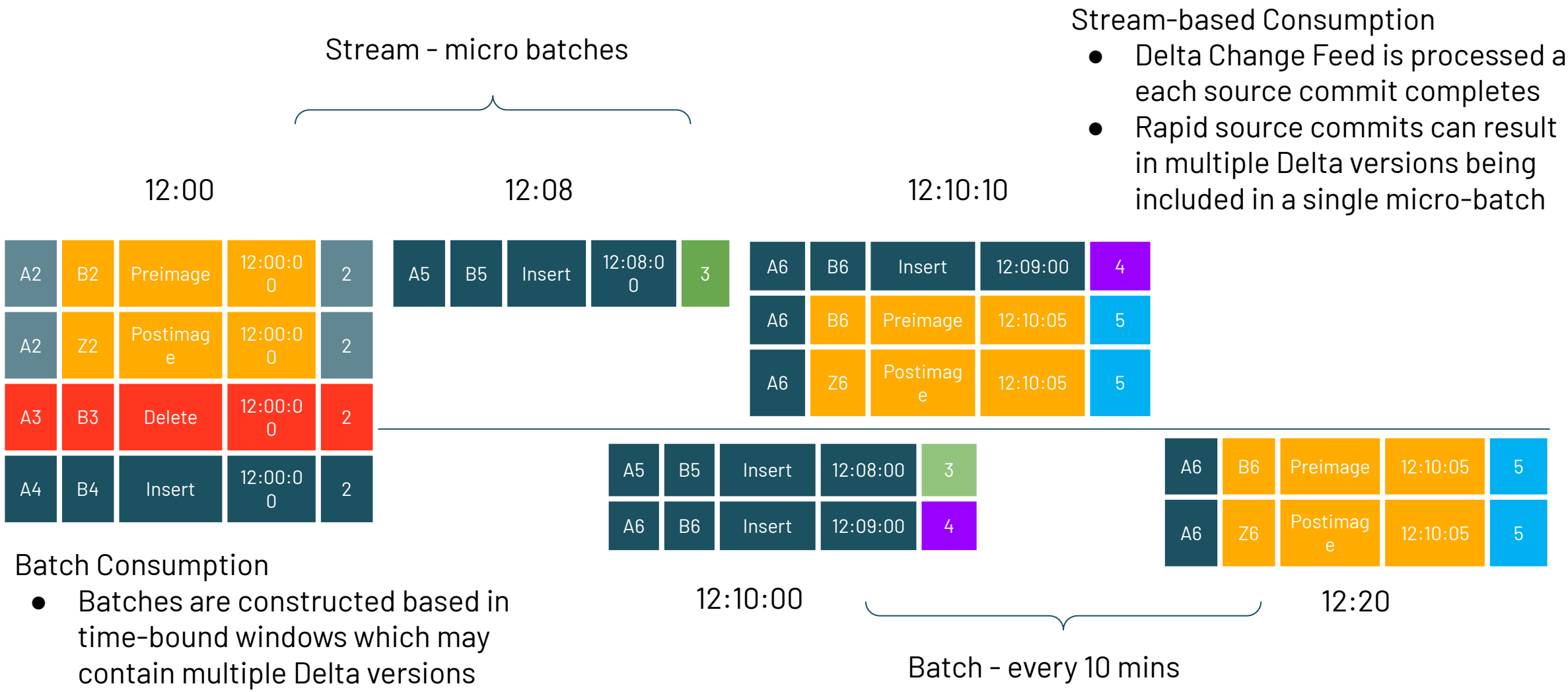


Change Data Feed Output

PK	B	Change Type	Time	Version
A2	B2	Preimage	12:00:00	2
A2	Z2	Postimage	12:00:00	2
A3	B3	Delete	12:00:00	2
A4	B4	Insert	12:00:00	2

A1 record did not receive an update or delete.
So it will not be output by CDF.

Consuming the Delta Change Data Feed



Stream-based Consumption

- Delta Change Feed is processed as each source commit completes
- Rapid source commits can result in multiple Delta versions being included in a single micro-batch

Batch Consumption

- Batches are constructed based in time-bound windows which may contain multiple Delta versions

Typical Use Cases

Silver & Gold Tables

Improve Delta performance by processing only changes following initial MERGE comparison to accelerate and simplify ETL/ELT operations

Materialized Views

Create up-to-date, aggregated views of information for use in BI and analytics without having to reprocess the full underlying tables, instead updating only where changes have come through

Transmit Changes

Send Change Data Feed to downstream systems such as Kafka or RDBMS that can use it to incrementally process in later stages of data pipelines

Audit Trail Table

Capturing Change Data Feed outputs as a Delta table provides perpetual storage and efficient query capability to see all changes over time, including when deletes occur and what updates were made

When to Use Delta Change Data Feed

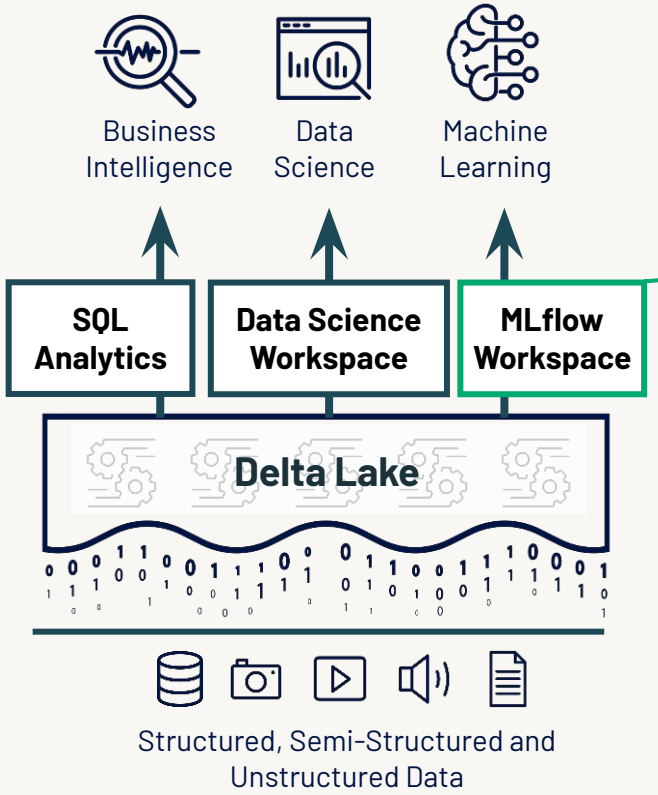
- Delta changes include updates and/or deletes
- Small fraction of records updated in each batch
- Data received from external sources is in CDC format
- Send data changes to downstream application

- Delta changes are append only
- Most records in the table updated in each batch
- Data received comprises destructive loads
- Find and ingest data outside of the Lakehouse

Advanced Topics

Deploying Streaming Models from the MLflow Model Registry

MLflow Model Registry in the Lakehouse

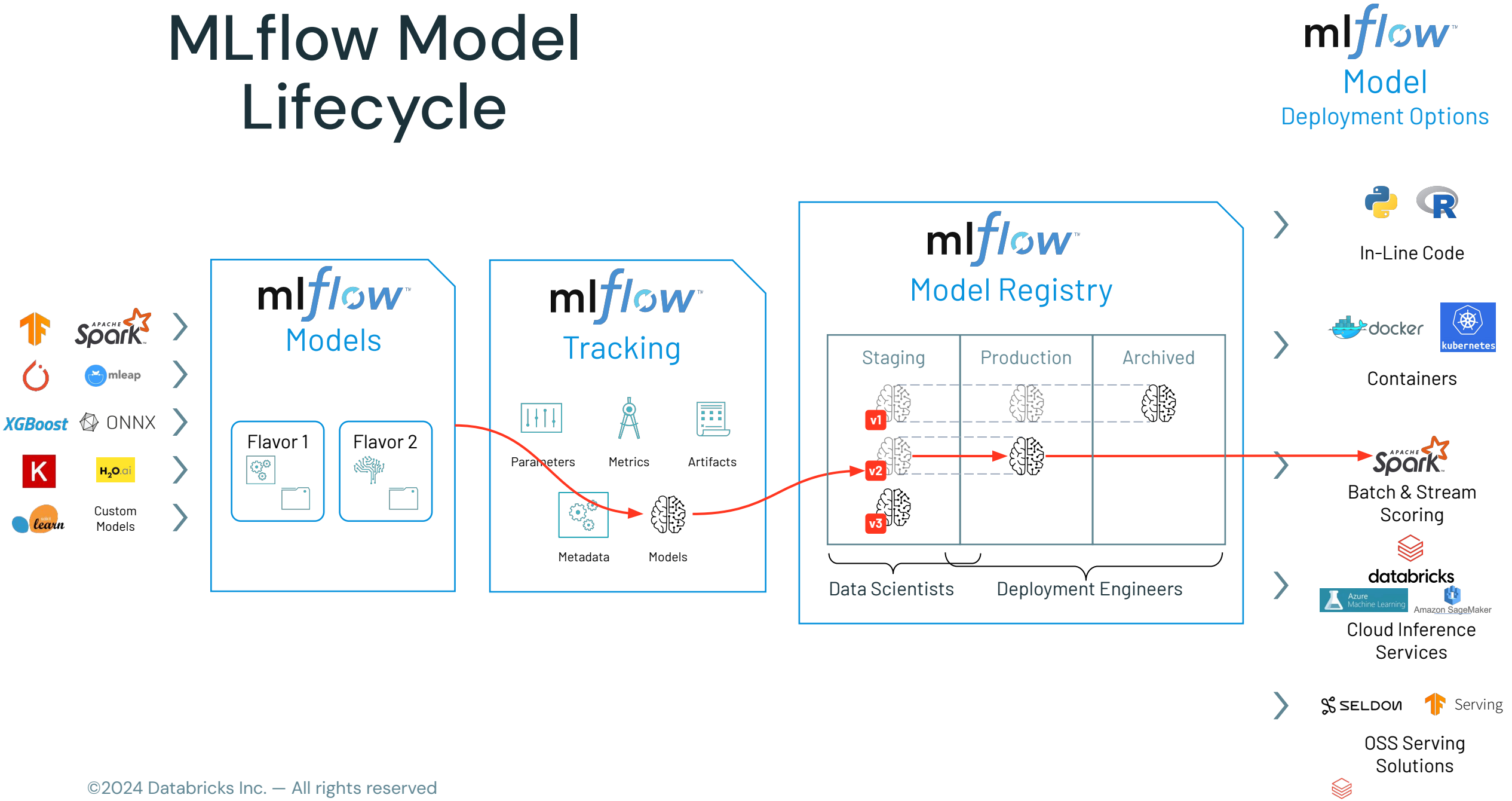


MLflow is the foundation for supporting the end-to-end ML Model Lifecycle in Databricks.

The **MLflow Model Registry** is a managed service that is part of the MLflow offering in Databricks.

Powered by mlflow

MLflow Model Lifecycle



Model deployment

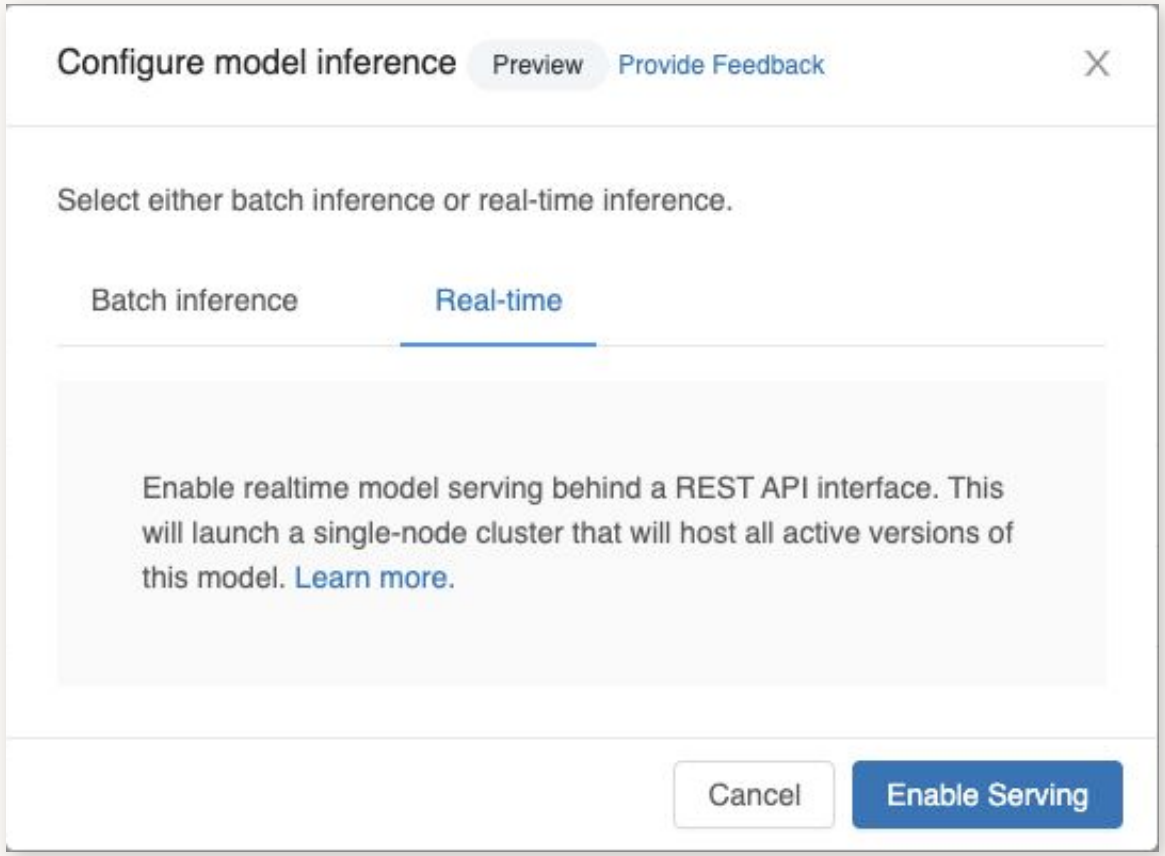
Flexible deployment at any scale

Batch scoring

One-click deployment of models from the Model Registry to scalable compute clusters for batch scoring

Online scoring

One-click deployment of models to REST endpoints for auto-scaling low latency scoring



Model deployment

Flexible deployment at any scale

As a UDF in a Streaming

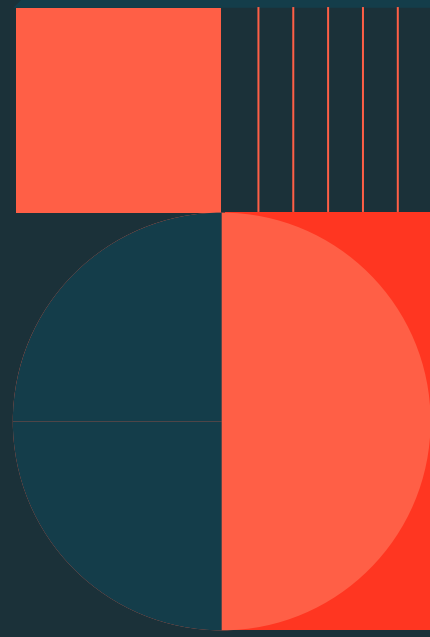
Deploy the scoring/inferencing as a pyfunc UDF (User Defined Function) and apply streaming data into the model.

```
import mlflow
loaded_model = mlflow.pyfunc.spark_udf(spark, model_uri=f"models:/...")


spark.readStream
     .table("silver_chicago")
     .select("id", "batch_date", "price", loaded_model(*columns).alias("predicted"))
     .writeStream.format("delta")
    [...]
```

09-streaming-mlflow

Production Best Practices



Streaming in Production

Things to Consider

- Unit Testing
- Triggers
- Stream Names
- Fault Tolerance
- State Management
- Multiplexing
- Monitoring & Instrumentation
- Application (Cost) Optimization
- Troubleshooting

Streaming in Production: Collected Best Practices, Part 1

Streaming in Production: Collected Best Practices, Part 2

Unit Testing

Before Deployment

cco

- Modularize Code
  - Divide code into testable chunks
  - Organize business logic into functions calling other functions.
  - Don’t code in dependencies on the global state or external systems
  - DataFrame in -> DataFrame out
- Test Everything
  - Databricks Unit Testing

cco

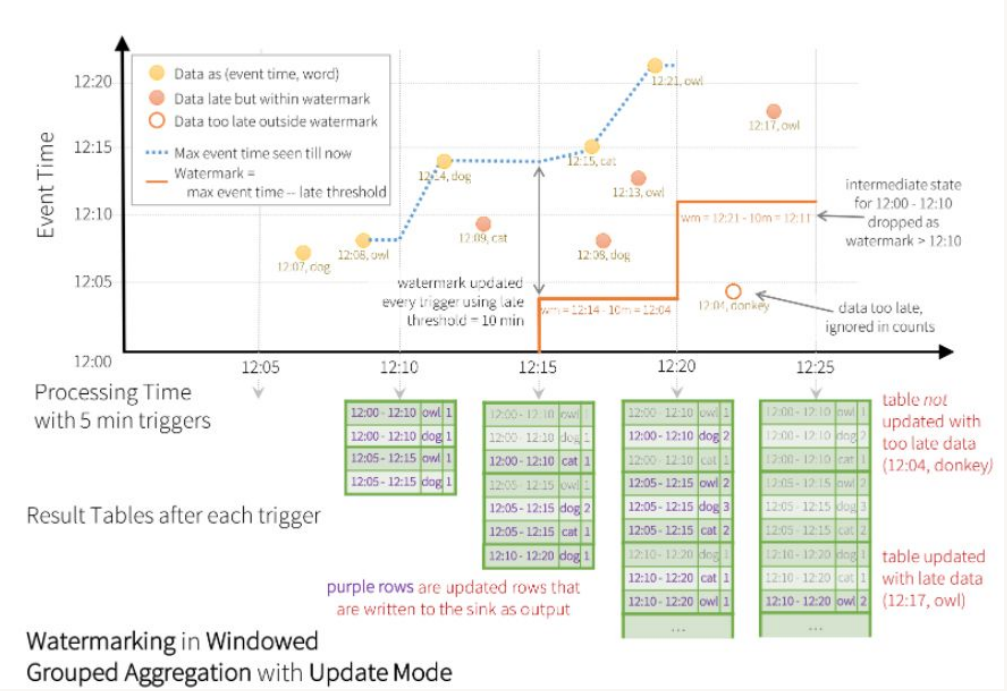
- Leverage Existing Libraries
  - built-in Spark test suite
  - spark-testing-base
  - spark-fast-tests (Scala)
  - chispa, nutter (Python)

Triggers

Before Deployment

Windows and Triggers
and Watermarks, Oh My!

- Is the stream always on or just a batch substitute?
- What are trigger intervals?
- SLAs?
- Trigger.Once/Trigger.AvailableNow



Name Your Stream

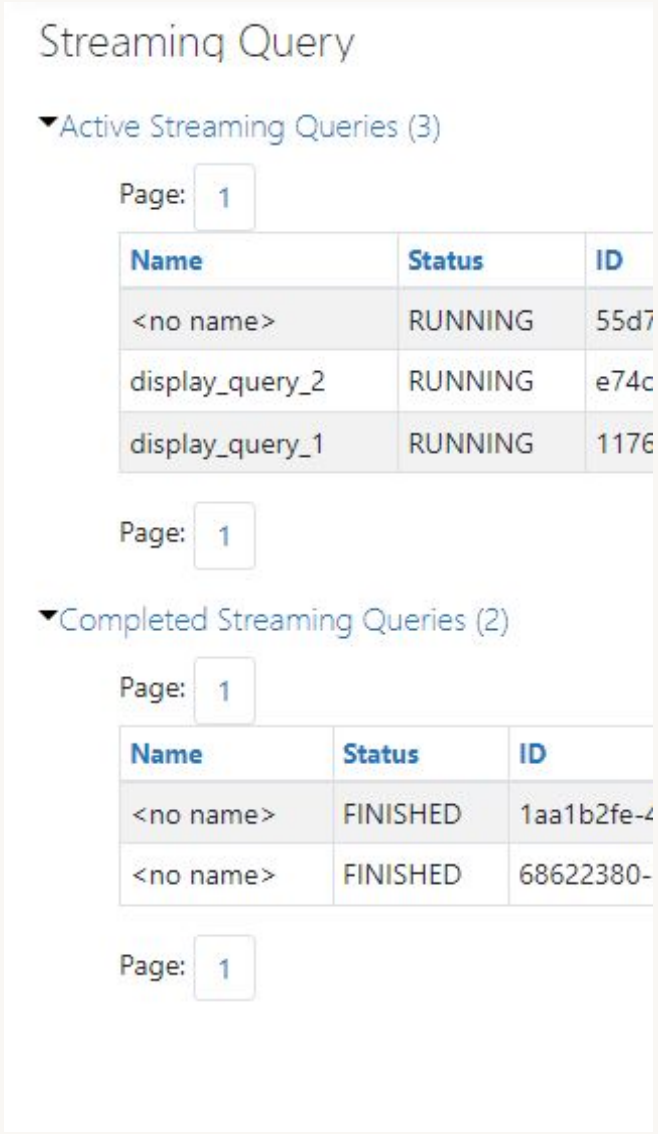
Before Deployment

We give books titles.

We give airline flights numbers.

We give streaming apps names.

```
(
streamingDataFrame
      .writeStream
      .format("delta")
      .option("path", "")
      .queryName("A_Named_Stream")
      .start()
)
```



Fault Tolerance

Before Deployment

ALWAYS

set

a

UNIQUE

checkpoint

for

EVERY

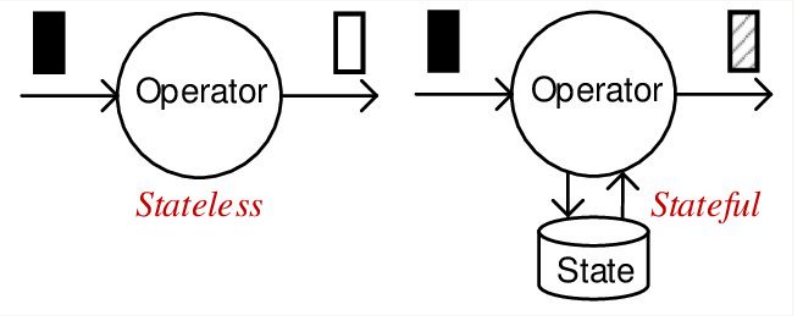
stream

```
(
streamingDataFrame
    .writeStream
    .format("delta")
    .option("path", "")
    .queryName("A_Named_Stream")
    .option("checkpointLocation",
            "<your_location>")
    .start()
)
```

State Management & RocksDB

Before Deployment

- Stateful Operations?
- High GC pressure = problem
- RocksDB will not make your stream faster
  - (usually, but maybe, sometimes)



Liu et al., 2020

```
spark.conf.set(
  "spark.sql.streaming.stateStore.providerClass",
  "com.databricks.sql.streaming.state.RocksDBStateStoreProvider")
```

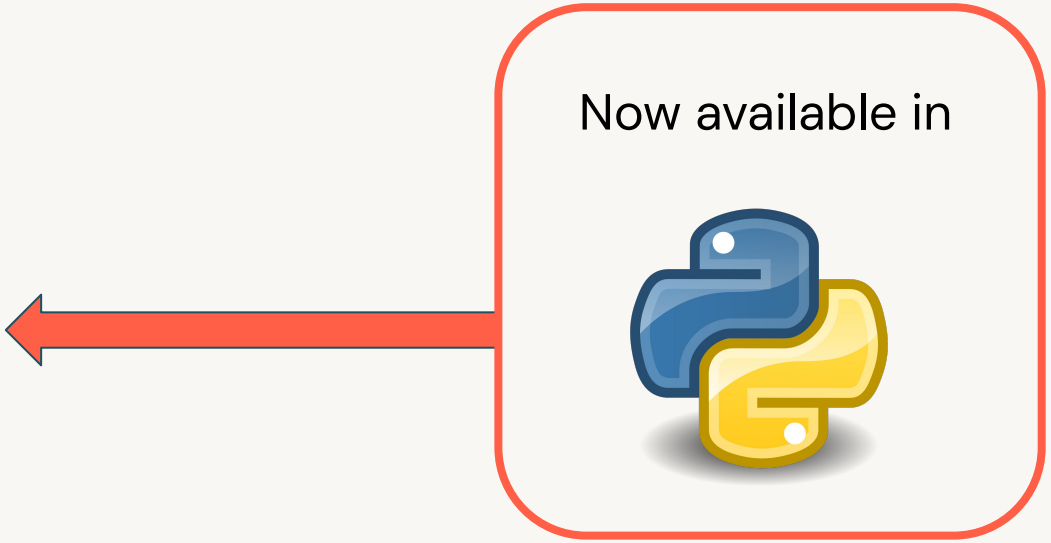
Meta Open Source

Monitoring & Instrumentation

After Deployment

Tooling Options

- Internal
  - SparkUI (streaming dashboard)
  - Notebook UI
  - `StreamingQueryListener` class
- External
  - Datadog
  - Build your own
    - Dropwizard, Grafana, Log Analytics, etc.
- "How to Monitor Streaming Queries in PySpark"



Python Software Foundation

Open Discussion – Bring your use case

Open Discussion – Bring your use case

Bring your use case

- Use case discovery and open discussion