# Einführung in MLOps

## 13 Stream Processing

Tobias Mérinat
teaching2025@fsck.ch

Lucerne University of
Applied Sciences and Arts
HOCHSCHULE
LUZERN

Department of Information Technology
Lucerne University of Applied Sciences and Arts
6343 Rotkreuz, Switzerland

14. und 15. Februar 2025

# Stream-Processing

- Auch Event-Stream-Processing genannt
- Ein- und Ausgabeobjekt von Berechnungen sind *Sequenzen von Ereignissen über die Zeit*
- Berechnung beginnt, sobald ein Ereignis eintritt

# Vergleich mit Batch-Processing

- Batch-Processing sammelt Ereignisse, Berechnung erfolgt periodisch
- Batch-Prozessierung ist effizient (Parallelität kann ausgenützt werden)
- Stream-Processing ist schnell (Verarbeitung direkt nach Eintreten)
- Stream-Processing ist effektiv (nur benötigte Berechnungen werden ausgeführt)

# Voraussetungen

- Streaming Transport (Kafka, Redpanda, AWS Kinesis, GCP Dataflow, . . . )
- Stream Computing Engine (Apache Flink, KSQL, Spark Streaming, Quix, . . . )

# Herausforderungen

- Stream Processing ist komplex
- Es unterscheidet sich fundamental von Batch Processing
- Arbeiten mit Streams erfordert einen mentalen Wandel
- Im folgenden einige Herausforderungen

# Aggregation, Reihenfolge, Duplikate

- Daten sind *unbounded*, es gibt keinen definierten Start und kein Ende
- Will man aggregieren, muss man *Windows* definieren
  - Tumbling, Hopping, Sliding, Sessions, Snapshot Windows
- Events können verspätet ankommen. Wie lange warten wir?
- Reihenfolge ist nicht garantiert
- Duplikate können auftreten

# Memory

- Aufgrund der variable Ankunftsraten können Memory-Spikes auftreten
- Aggregationen können lange laufen, ganzes Window muss im Speicher gehalten werden
- Speicherverbrauch ist demnach abhängig von
  - der Grösse des Windows,
  - der Event-Frequenz
  - der Body Size

# Komplexität und Tradeoffs

- Die Wahl der Fensterart und -grösse
- Features sind per Def. erst dann akkurat, wenn das ganze Fenster gefüllt ist
- Tradeoff zwischen Aktualität, Genauigkeit, Memory und CPU
- Joins über Streams und Zeit sind komplex

# CAP Theorem

- Das CAP Theorem sagt aus:
  - In verteilten Systemen besteht ein Tradeoff zwischen Verfügbarkeit und Konsistenz
  - Nur zwei von *Consistency*, *Availability* und *Partition Tolerance* können durchgehend garantiert werden

# Was die Adoption von Stream Processing bremst

Gemäss einer Reihe von Interviews, durchgeführt von Chip Huyen, sind die Gründe, warum Firmen zögern, Stream Processing Fähigkeiten aufzubauen, die folgenden:

- Firmen sehen keinen Vorteil, da
  - ihre Infra nicht gross genug ist, dass Inter-Service-Kommunikation zum Flaschenhals wird
  - sie keine Applikationen haben, welche von Online-Prediction profitieren würde
  - sie zwar solche haben, ihnen dies aber nicht bewusst ist, da Erfahrung in diesem Gebiet fehlt
- Hohe initiale Kosten für neue Infrastruktur
- Hohe laufende Kosten (im Gegensatz zur Batch-Verarbeitung)
- Neben Python auch Java/Scala Knowhow notwendig (Kafka, Flink, weitere)

Streaming-Technologien haben in den vergangenen paar Jahren jedoch rasante Fortschritte gemacht, welche die drei letztgenannten Punkte zum Teil entkräften.

# Konzepte - Data Flow

- Jede Event Notification, also jeder Datenpunkt, fliesst durch eine Serie von Tranformationen
- Ein solcher Data Flow besteht aus
  - Source: Applikation, Sensor, Change Data Capture log, Social Media Feed, IoT Device, Queue
  - Transformationen: anreichern, filtern, aggregieren, modifizieren
  - Sink: Datenbank, Open Table Format, Queue, API, . . .

# Konzepte - State

- Bei jeder Operation, welche mehrere Rows (Events) verwendet, muss der Cluster einen *State* speichern können, z.B. bei:
  - Aggregationen über Windows
  - group by
  - joins

# Konzepte - Zeitstempel und Watermarking

Jede Event Notification hat zwei Arten von Zeitstempel:

- **Event Time**: Wann der Event passiert ist (vom Quellsystem aufgezeichnet)
- **Processing Time**: Wann die Event Notification verarbeitet wurde (vom verarbeitenden System aufgezeichnet)

Watermarking:

- Wird die Zeit zwischen Event Time und Processing Time zu gross, hat sich der Event verspätet
- Mittels einem *Watermark* definieren wir, wie lange nach dem Ende eines Fensters wir noch auch verspätete Event warten

# Konzepte - Backpressure

- Die Quelle produziert schneller, als die Event Notifications verarbeitet werden können
- Das Memory füllt sich langsam
- Dies wird *Backpressure* genannt

# Konzepte - Joins

- Standard Joins joinen auf eine ID, ohne constraint auf einen Zeitstempel
- Time-Based Joins verwenden zusätzlich den Aspekt Zeit, um zu joinen
- Join-Arten: *Interval Joins*, *Temporal Joins*, *Lookup Joins* und *Window Joins*
- Dabei geht es jeweils darum, den State (Memory-Verbrauch) einzuschränken und doch den Output zeitnah zur Verfügung zu stellen

# Konzepte für die Übungen

- **Producer** und **Consumer**: Erstere senden, zweitere lesen Messages
- **Topics**: Ein Thema, Consumer abonnieren Themen (*die Autobahn von Bern nach Zürich*)
- **Partition**: Unterteilung eines Topics (*Spur mit durchgezogener Linie*)
- **Key** einer Message: Pinnt diese an eine Partition, so können sich Messages nicht überholen
- **Consumer Group**: Teilnehmer einer Consumer Group lesen das gleiche Topic, aber von separaten Partitionen. Erhalten somit nicht die gleichen Events. Verschiedene Consumer Groups können das gleiche Topic lesen (Messages werden so mehrfach gelesen).