Inhaltsverzeichnis

1 Data Engineering Storage Abstractions

1.1 Data Lake

- Ein Data Lake ist ein zentrales Repository, das strukturierte, semi-strukturierte und unstrukturierte Rohdaten in großem Umfang speichert
- Eignet sich besonders für Big-Data-Analysen und Machine Learning, erfordert aber zusätzliche Verarbeitung zur Datennutzung
- Beispiel: Amazon S3

1.2 Data Warehouse

- Ein Data Warehouse ist ein zentrales, strukturiertes System zur langfristigen Speicherung und Analyse großer Datenmengen aus verschiedenen Quellen
- Optimiert für komplexe Abfragen und Business-Intelligence-Anwendungen
- Beispiel: MS SQL Server

1.3 Cloud Data Warehouse

- Ein Cloud Data Warehouse ist ein Data Warehouse, das in der Cloud gehostet wird (keine eigene Infrastruktur notwendig) und dadurch flexibel skalierbar und wartungsarm ist
- Beispiele: Snowflake, Google BigQuery oder Amazon Redshift

1.4 Data Lakehouse

- Datenarchitektur, die die Flexibilität und Skalierbarkeit eines Data Lakes mit den strukturierten Datenmanagement- und Analysefunktionen eines Data Warehouses kombiniert
- Ermöglicht sowohl explorative Datenanalyse als auch strukturierte BI-Abfragen auf einer gemeinsamen Plattform
- ACID Transaktionen:
  - Atomicity: Transaktion wird entweder vollständig ausgeführt oder gar nicht – es gibt keine halbfertigen Zustände
  - Consistency: Transaktion überführt die Datenbank von einem konsistenten Zustand in einen anderen, wobei alle definierten Regeln (z.B. Integritätsbedingungen) eingehalten werden
  - Isolation: Gleichzeitige Transaktionen beeinflussen sich nicht gegenseitig – jede Transaktion läuft so, als wäre sie allein im System
  - Durability: Sobald eine Transaktion abgeschlossen ist, bleiben ihre Änderungen dauerhaft gespeichert, selbst bei Systemausfällen
- Beispiel: Databricks

1.5 Delta Lake

- Open-Source-Storage-Schicht, die auf Data Lakes (z.B. in S3 oder Azure Blob Storage) aufsetzt und Funktionen wie ACID-Transaktionen, Schema-Management, Zeitreisen (Versionierung) und verlässliche Upserts/Merges bietet
- Damit macht Delta Lake rohe Data Lakes zuverlässiger, konsistenter und besser für Analytics und Machine Learning geeignet – indem es typische Probleme von Data Lakes (wie inkonsistente Daten oder fehlende Transaktionen) löst. Es wird oft mit Apache Spark und Databricks genutzt

- Schema Enforcement:
  - Stellt sicher, dass neue Daten, die in eine Delta-Tabelle geschrieben werden, dem bestehenden Tabellenschema entsprechen
  - Dadurch werden fehlerhafte oder inkonsistente Daten (z.B. falsche Datentypen oder fehlende Spalten) automatisch abgelehnt, um Datenqualität und Konsistenz zu gewährleisten
- Time Travel:
  - Ermöglicht den Zugriff auf frühere Versionen einer Tabelle, indem man einen bestimmten Zeitpunkt oder Versionsnummer angibt
  - Historische Daten analysieren, vergleichen, Wiederherstellungen durchführen oder versehentliche Änderungen rückgängig machen
- Beispiel: Delta Lake on Databricks

1.6 Data Mart

- Spezialisierte Teilmenge eines Data Warehouses, die auf die Anforderungen eines bestimmten Fachbereichs zugeschnitten ist
- Verbessert die Performance und Übersichtlichkeit für gezielte Analysen
- Beispiel: Amazon Redshift Data Mart

1.7 Data Mesh

- Dezentraler Ansatz zur Datenarchitektur, bei dem einzelne Teams für ihre eigenen Datenprodukte verantwortlich sind ("Data as a Product")
- Fördert Skalierbarkeit und Eigenverantwortung durch domänenorientierte Datenverwaltung

1.8 Data Vault

- Modellgetriebene Methode zur Gestaltung von Data Warehouses, die speziell für historisierbare, skalierbare und auditierbare Datenarchitekturen entwickelt wurde
- Sie trennt Daten in drei Hauptkomponenten:
  - Hubs (Schlüsselentitäten, z.B. Kunden-ID),
  - Links (Beziehungen zwischen Hubs, z.B. Kunden $\leftrightarrow$ Bestellungen)
  - Satellites (beschreibende, historisierte Daten mit Zeitstempel, z.B. Kundenname, Adresse).
- Eignet sich besonders für agile, stark wachsende Datenumgebungen mit hohem Bedarf an Nachvollziehbarkeit und Flexibilität.

2 Datenmanagement

- Übergeordnete Prozess der Erfassung, Speicherung, Organisation, Pflege und Nutzung von Daten in einem Unternehmen
- Umfasst verschiedene Disziplinen wie Data Governance, Datenintegration, Datenqualität, Metadatenmanagement und Archivierung, um den wertschöpfenden Einsatz von Daten sicherzustellen
- Datenintegrität: Zusammenführen und Vereinheitlichung von Daten aus verschiedenen Quellen

2.1 Data Governance

- Bezeichnet den Rahmen aus Richtlinien, Prozessen und Verantwortlichkeiten, der sicherstellt, dass Daten im Unternehmen korrekt, sicher, einheitlich und regelkonform verwaltet werden
- Umfasst Themen wie Datenqualität, Datenschutz, Zugriffsrechte, Compliance und Datenverantwortung, um Vertrauen und Kontrolle über Daten zu gewährleisten

2.2 Datenqualität

- Vollständigkeit → Sind alle erforderlichen Datenfelder vorhanden und ausgefüllt?
- Korrektheit → Entsprechen die Daten den realen, erwarteten Werten (z.B. stimmen Postleitzahlen mit Städten überein)
- Konsistenz → Sind die Daten in verschiedenen Systemen oder Tabellen widerspruchsfrei (z.B. gleicher Kundennamen in allen Datensätzen)
- Aktualität → Sind die Daten aktuell bzw. zeitgerecht verarbeitet (z.B. keine veralteten Transaktionen im Reporting)
- Eindeutigkeit → Gibt es doppelte Datensätze, wo es keine geben sollte (z.B. doppelte Kunden-IDs)
- Validität → Entsprechen die Daten den erwarteten Formaten oder Regeln (z.B. E-Mail-Adressen im gültigen Format, Zahlen in numerischen Feldern)

2.3 Datenintegration

- Prozess, bei dem Daten aus verschiedenen Quellen zusammengeführt, vereinheitlicht und für eine zentrale Nutzung bereitgestellt werden
- Ziel ist es, konsistente, vollständige und aktuelle Informationen für Analysen, Berichte oder operative Systeme bereitzustellen

2.4 Daten Lifecycle

- Beschreibt die verschiedenen Phasen, die Daten während ihrer Existenz durchlaufen – von der Erfassung über Speicherung, Verarbeitung, Nutzung und Weitergabe bis hin zur Archivierung oder Löschung
- Hilft, Daten systematisch zu verwalten, deren Qualität zu sichern und rechtliche sowie sicherheitsrelevante Anforderungen zu erfüllen

2.5 Data Catalog

- Zentrales Verzeichnis, das Metadaten über Datenbestände innerhalb eines Unternehmens organisiert, beschreibt und auffindbar macht

- Hilft Nutzern dabei, Datenquellen schnell zu finden, deren Struktur und Bedeutung zu verstehen und die Daten effizient zu nutzen – oft unterstützt durch Suchfunktionen, Tags und Datenklassifizierung

2.6 Unity Catalog

- Einheitliches Datenverwaltungssystem von Databricks, das Zugriffskontrolle, Daten-Governance und Katalogisierung für alle Daten-Assets über verschiedene Workloads hinweg (z.B. SQL, Python, BI-Tools) ermöglicht
- Bietet zentrale Verwaltung von Benutzerrechten, Datenklassifikation und Lineage (Herkunftsnachverfolgung) über mehrere Workspaces und Clouds hinweg – für mehr Sicherheit, Transparenz und Zusammenarbeit in Data- und ML-Projekten

2.7 Feature Store

- Zentrale Plattform zur Speicherung, Verwaltung und Wiederverwendung von Merkmalen (Features), die für Machine-Learning-Modelle verwendet werden
- Ermöglicht konsistente und effiziente Bereitstellung von Features sowohl für das Training als auch für die Echtzeit-Vorhersage in produktiven ML-Systemen

3 Daten

3.1 Datenprodukt

- Wiederverwendbares, klar definiertes Datenartefakt (Datensatz), das für einen konkreten Anwendungsfall erstellt wird und von anderen leicht konsumiert werden kann

3.2 Data Lineage

- Beschreibt die Herkunft, den Weg und die Transformationen von Daten über verschiedene Systeme hinweg – vom Ursprung bis zur Nutzung, etwa in Berichten oder Analysen
- Hilft dabei, Datenflüsse nachvollziehbar zu machen, die Datenqualität zu sichern und regulatorische Anforderungen zu erfüllen

3.3 Daten Versionierung

- Systematische Erfassen, Speichern und Verwalten verschiedener Zustände oder Versionen von Datensätzen im Zeitverlauf
- Dadurch können Änderungen nachvollzogen, frühere Datenzustände wiederhergestellt und Reproduzierbarkeit in Analysen und Machine-Learning-Modellen sichergestellt werden

3.4 Datenmodellierung

- Prozess der strukturierten Darstellung von Daten und deren Beziehungen untereinander, meist als Diagramm oder Modell
- Ziel ist es, ein klares, logisches Gerüst für die Speicherung, Verwaltung und Nutzung von Daten zu schaffen – typischerweise in Form von konzeptionellen, logischen und physischen Datenmodellen

3.4.1 Methoden

- Relationales Modell:
  - Grundlage für relationale Datenbanken; Daten werden in Tabellen (Relationen) mit Zeilen und Spalten organisiert
  - Beispiel: Tabelle Kunde mit Spalten wie KundenID, Name, Adresse.
- Entity-Relationship-Modell:
  - Visualisiert Entitäten und deren Beziehungen zur konzeptuellen Planung von Datenstrukturen
  - Beispiel: Entitäten *Kunde* und *Bestellung* sind über eine Beziehung *tätigt* verbunden
- Star Schema:
  - Häufig im Data-Warehouse-Bereich; eine zentrale Faktentabelle ist mit mehreren Dimensionstabellen verbunden
  - Beispiel: Faktentabelle *Umsatz* mit Verbindungen zu *Produkt, Zeit, Kunde, Standort*

3.5 Datenformate

- Parquet:
  - Spaltenbasiertes Speicherformat
  - Komprimiert und effizient für Analyse-Workloads

- CSV:
  - Einfaches, textbasiertes Format
  - Weit verbreitet, aber keine Schema- oder Typinformation
- JSON:
  - Textbasiert, semi-strukturiert
  - Gut für hierarchische Daten, aber größer und langsamer als binäre Formate
- Delta Table:
  - Baut auf dem Apache Parquet Format auf und wird durch Delta Lake erweitert, um transaktionale Konsistenz, Schema-Evolution und Zeitreisen (Versionierung) in Data Lakes zu ermöglichen.
  - Erlaubt ACID-Transaktionen auf großen Datenmengen, was insbesondere bei Big-Data-Analysen und Machine Learning für zuverlässige, reproduzierbare Datenpipelines sorgt
- XML:
  - Textbasiert, hierarchisch, aber oft sehr groß und weniger performant

3.6 Feature Engineering

- Skalierbare und automatisierte Aufbereitung von Merkmalen (Features) für Machine-Learning-Modelle innerhalb von Datenpipelines
- Data Engineers bauen dafür robuste, reproduzierbare Prozesse zur Berechnung, Speicherung und Bereitstellung von Features – z.B. über Feature Stores – und sorgen dafür, dass diese Merkmale konsistent, versioniert und performant für Training und Inferenz verfügbar sind

4 Daten Pipeline

4.1 ETL

- Prozess, bei dem Daten aus verschiedenen Quellen extrahiert, in ein geeignetes Format transformiert und schließlich in ein Zielsystem wie ein Data Warehouse geladen werden
- Dient dazu, Rohdaten für Analysen nutzbar zu machen, indem sie bereinigt, vereinheitlicht und angereichert werden
- Moderne ETL-Prozesse können auch in Echtzeit (Streaming-ETL) oder als ELT-Variante ablaufen, bei der die Transformation nach dem Laden erfolgt

4.2 ETL Pipeline

- Extraktion → ADF (Extrahieren von Daten aus verschiedenen Quellen)
- Transform → Databricks und Spark (Verarbeiten, bereinigen und anreichern der Daten in skalierbaren Umgebungen)
- Load → Snowflake, BigQuery (Laden der transformierten Daten in ein Cloud DWH)
- Orchestrierung → Apache Airflow (Steuerung und Automatisierung des Ablaufs der gesamten ETL-Prozesse)
- Montoring → Grafana, Azure Monitor (Überwachung von Performance, Ausfällen und Fehlern der Pipeline)
- Data Quality & Testing → Great Expectations (Validieren der Datenqualität durch automatisierte Tests und Regeln)

4.3 Pipeline Montoring

- Überwachung von Datenpipelines, Systemen und Prozessen, um sicherzustellen, dass Daten zuverlässig, korrekt und zielgerecht verarbeitet werden
- Fehlererkennung:
  - Erkennen von fehlgeschlagenen ETL-Jobs
  - Identifikation von Datenanomalien
  - Monitoring von Datenlatenzen
- Performance Überwachung:
  - Laufzeit von Pipelines
  - Ressourcenverbrauch
  - Datenvolumen und Verarbeitungsraten
- Sicherstellen der Datenqualität:
  - Validierung von Daten gegen Regeln
  - Schema Überwachung
- Transparenz und Reporting:
  - Dashboarding von Metriken (Grafana, Power BI)
  - Alerting bei Schwellenwertüberschreitung
- Beispiele:
  - Apache Airflow Monitoring: Überwacht den Status von DAGs, ob Tasks fehlschlagen oder hängenbleiben
  - Snowflake oder BigQuery Monitoring: Überwachung von Query-Ausführungszeiten und -Kosten

  - Streaming Monitoring: Sicherstellen, dass Messages rechtzeitig und vollständig verarbeitet werden

4.4 CI/CD

- CI: Automatisches Testen und Bauen bei jeder Codeänderung
- CDelivery: Automatisches Ausliefern in eine Staging-Umgebung
- CDeployment: Automatisches Ausliefern bis in die Produktion (ohne manuelle Eingriffe)

4.5 Orchestrierung

- Zentrale Steuerung, Planung und Überwachung von Datenprozessen (wie ETL- oder ELT-Jobs), sodass sie in der richtigen Reihenfolge, zur richtigen Zeit und abhängig voneinander ausgeführt werden
- Sorgt für Automatisierung, Fehlerbehandlung, Wiederholbarkeit und Effizienz komplexer Datenpipelines – typischerweise mit Tools wie Apache Airflow, Prefect oder Dagster

4.6 Logging

- Strukturiertes Erfassen und Speichern von Informationen über den Ablauf, Status und mögliche Fehler von Datenprozessen (z.B. ETL-Jobs, Pipelines, APIs)
- Dient der Überwachung, Fehlerdiagnose, Auditierbarkeit und Performanceanalyse von Datenflüssen – oft zentral für Debugging und Betriebssicherheit

4.7 Data Analytics Konzepte

- EDA
- Deskriptive, diagnostische, prädiktive und präskriptive Analysen
- KPI-Tracking: Überwachung von Leistungskennzahlen

5 Dokumentation

- Datenquellen:
  - Herkunft (Systeme, APIs, Dateien)
  - Zugriffsmethoden und -rechte
  - Aktualisierungsfrequenz
- ETL-/ELT-Prozesse:
  - Datenflüsse und Pipeline-Übersichten
  - Verwendete Transformationslogik (SQL, dbt, Spark etc.)
  - Abhängigkeiten zwischen Schritten
- Datenmodelle und Schemata:
  - Tabellen, Views, Spaltenbeschreibungen
  - Beziehungen zwischen Entitäten (z.B. ER-Diagramme)
  - Versionierung von Modellen
- Datenqualitätsregeln:
  - Validierungschecks
  - Testdefinitionen (z.B. dbt tests, Great Expectations)
- Zugriffs- und Sicherheitsrichtlinien:
  - Rollen und Berechtigungen
  - Datenklassifizierung (z.B. PII, öffentlich, vertraulich)
- Monitoring & Alerting:
  - Überwachte Metriken
  - Fehlerbehandlung und Wiederanläufe
  - SLAs und SLOs
- Code- und Tool-Dokumentation:
  - Repos, Technologien, Abhängigkeiten
  - Setup-Anleitungen und Deployment-Prozesse

6 Batch vs. Streaming Processing

6.1 Batch Processing

- Verarbeitet statische Datenmengen in festen Zeitabständen oder auf Abruf
- Daten liegen vollständig vor, bevor sie verarbeitet werden
- Höhere Latenz, dafür oft einfachere Verarbeitung und Fehlerbehandlung
- Gut geeignet für regelmäßige Reports, ETL-Prozesse, Backfills
- Beispiele: Spark Batch, Hadoop MapReduce, SQL-Jobs

6.2 Wann Batch Processing besser ist

- Arbeiten mit großen Datenmengen in regelmäßigen Abständen (z.B. täglich, stündlich)
- Echtzeitreaktion nicht notwendig ist (z.B. Reporting, Data Warehousing, Archivierung)
- Komplexe Analysen oder Aggregationen auf vollständigen Datensätzen
- Datenqualität, Wiederholbarkeit und Genauigkeit wichtiger sind als Geschwindigkeit

6.3 Streaming Processing

- Verarbeitet Daten kontinuierlich und nahezu in Echtzeit
- Daten werden direkt bei Eintreffen verarbeitet
- Geringe Latenz, dafür komplexer in Fehlerhandling und Zustandsverwaltung
- Ideal für Echtzeitanalysen, Monitoring, Betrugserkennung, IoT
- Beispiele: Spark Structured Streaming, Apache Flink, Kafka Streams

6.4 Wann Streaming Processing besser ist

- Auf Ereignisse muss sofort reagiert werden (z.B. Alarme, Benachrichtigungen, Live-Metriken)
- Daten kontinuierlich einlaufen (z.B. Logdaten, Sensoren, Webtraffic)
- Niedrige Latenz (Echtzeit oder Near-Real-Time)
- Laufendes Überwachen oder Steuern von Systemen

7 Spark

7.1 Basics

- Open Source computing engine for parallel data processing on computer clusters
- Cluster of computers pools the resources of many machines together to use all cumulative resources as if they were a single computer
- A Spark application is submitted to these cluster managers which will grant resources to the application

7.2 Spark Application

- Spark applications consists of a driver process and a set of executor processes
- Driver:
  - Creates SparkContext and SparkSession
  - Runs the `main()` function, sits on a node in the cluster, and is responsible for three things:
    - Maintaining information about the Spark application
    - Responding to a user's program or input
    - Analyzing, distributing, and scheduling work across the executors
  - The Driver process is the heart of the Spark application and maintains all relevant information during the lifetime of the application
- Executor (on worker node):
  - Responsible for carrying out the work that the driver assigns them
  - Each executor is responsible for 2 things:
    - Executing code in parallel (one per executor core) assigned to it by the driver
    - Reporting the state of computation on that executor back to the driver

7.3 Lazy Evaluation

- Spark waits until the very last moment to execute the graph (DAG) of computation instructions
- In Spark, a plan of transformations (DAG) is built up that is to be applied to the data rather than modifying the data immediately
- By waiting, Spark compiles this plan from the raw DataFrame transformations to a streamlined physical plan (DAG) that will run as efficiently as possible across the cluster
- As result, Spark can optimize the entire data flow from end to end (predicate pushdown)

7.4 DAG

- Logical representation of a Spark computation
- Consists of a series of transformations (narrow/wide) applied to a DataFrame
- Ensures that computations are structured efficiently and executed in the right order

7.5 Transformation

- Operation creating a new DataFrame from an existing one
- Transformations = logical transformation plan (DAG)
- Narrow transformations:
  - Operation where each partition is used by at most one partition in the output
  - No data shuffling between nodes, thus more efficient (e.g., `filter()`)

  - Executed in a single stage
- Wide Transformations:
  - Operation where data from one partition is used by multiple partitions in the output
  - Require data shuffling across nodes, thus expensive (e.g., `groupBy()`)
  - Create new stage in the DAG

7.6 Action

- Action = triggers the computation
- An action instructs Spark to compute a result from a series of transformations (e.g., `count()`)
- Every action starts a new job

7.7 Job

- Series of parallel transformations and actions (complete computations) performed on a DataFrame, triggered by an action
- Operate on partitions of the DataFrame
- Consists of:
  - Stages → each job is divided into multiple stages (set of operations)
  - Tasks → each stage consists of multiple tasks running in parallel on executor cores

7.8 Stage

- Collection of parallel tasks separated by shuffle boundaries
- Narrow transformations can be executed in the same stage
- Wide transformations require data shuffling and create new stages

7.9 Task

- Each Task runs on and processes one partition of the data

7.10 Partition

- To allow every executor to perform work in parallel, Spark breaks up the data into partitions
- Partition: Collection of rows that sit on one physical machine in the cluster
- Represents how the data is physically distributed across the cluster of machines during execution
- One partition = parallelism of one
- Many partitions but only one executor = parallelism of one

7.11 Summary

- SparkContext:
  - Core entry point for Spark operations
  - Created automatically inside a SparkSession
- SparkSession:
  - Entry point for a Spark application
  - Provides access to DataFrames, datasets, and SQL functionality
  - Created using: `SparkSession.builder.appName().getOrCreate()`
- Cluster:
  - Group of interconnected machines (nodes) working together to process large-scale data in parallel
  - Managed by cluster manager like YARN, Kubernetes, or Mesos
- Node:
  - Virtual or physical machine in the cluster
- Driver node:
  - Virtual or physical machine where the driver runs
  - Central coordinator of a Spark application
  - Creates SparkContext, schedules tasks, and collects results from executors
  - Allocates resources to executors
  - Converts Spark code into a DAG of tasks
- Driver core:
  - Controls how many CPU cores are allocated to the Spark driver for managing the Spark application and scheduling tasks
- Driver memory:
  - RAM allocated to the Spark driver for storing metadata, job information, and small collected results
- Worker node:
  - Virtual or physical machine in the cluster that contains and runs multiple executors to process data in parallel
- Executor:
  - Process running on a worker node
  - Responsible for executing tasks and storing intermediate data in memory or disk
- Executor core:
  - Number of CPU cores allocated to each executor
  - Defines how many Tasks an executor can run in parallel
- Executor memory:
  - RAM allocated to each executor for performing computations and storing cached data

7.12 Spark vs. Python

7.12.1 Python

- Für kleine bis mittelgroße Datenmengen, die in den lokalen Speicher passen
- Bei schnellen Analysen, Exploration und Prototyping
- Wenn viele Bibliotheken (z.B. Pandas, NumPy, scikit-learn) gebraucht werden
- Ideal für lokale Entwicklungsumgebungen und Skripting

7.12.2 Spark

- Bei sehr großen Datenmengen, die nicht in den Arbeitsspeicher passen
- Wenn Daten verteilt verarbeitet oder aus verteilten Quellen gelesen werden sollen
- Für skalierbare ETL-Prozesse, Datenpipelines oder wiederkehrende Batch-Jobs
- In Cluster-Umgebungen (z.B. Databricks, EMR, Kubernetes) mit mehreren Knoten

8 Clean Code

- Aussagekräftige Namen verwenden: Variablen, Funktionen und Klassen sollten klar und präzise beschreiben, was sie tun
- Funktionen klein und fokussiert halten: Jede Funktion sollte nur eine Aufgabe erfüllen, z.B. eine Transformation in Spark oder eine bestimmte Berechnung
- Code modularisieren und wiederverwenden: Wiederkehrende Logik in Funktionen oder Klassen auslagern, um Duplikate zu vermeiden (z.B. wiederverwendbare UDFs in Spark)
- Kommentare und Docstrings nutzen: Komplexe Logik mit kurzen Kommentaren erklären und Funktionen mit Docstrings dokumentieren (speziell bei Transformationen)
- Vermeiden magischer Zahlen und Strings: Statt harte Werte direkt im Code zu nutzen, Konstanten definieren und gut benennen
- Fehlerbehandlung sinnvoll einbauen: Ausnahmen abfangen, aber nicht still ignorieren; klare Fehlermeldungen geben
- DataFrame-Operationen lesbar schreiben: Ketten von Spark-Transformationen klar strukturieren, z.B. jede Operation in einer neuen Zeile
- Vermeiden zu komplexer Verschachtelungen: Schreibe lieber mehrere einfache Schritte als einen langen komplexen Block
- Teste kritische Transformationen: Nutzen von Unit-Tests oder kleine Integrationstests, um Datenqualität sicherzustellen (z.B. mit pytest und Spark Testing Base)