

5

Scalable DataOps for Machine Learning

Topics



Understanding the components of a scalable DataOps pipeline



Best practices and tools for designing and orchestrating scalable pipelines



Integrating machine learning into these pipelines



Advanced topics such as serverless computing and hybrid cloud environments



Summary

Path of Data Eng.





“

What is DataOps?

What is DataOps?

- **Introduction to Cron Tools**

- Cron tools are utilities used to schedule and automate repetitive tasks on Unix-like operating systems.
- Example: Crontab.guru is a popular tool for generating cron schedule expressions.
- Usage: Commonly used for tasks like backups, system maintenance, and script automation.

What is DataOps?

Limitations of Cron Tools

- **Basic Scheduling**
 - Suitable for simple task automation but not for complex data processing.
- **Scalability Issues**
 - Not designed to handle large-scale data operations or distributed systems.
- **Lack of Fault Tolerance**
 - No built-in mechanisms for handling job failures or node failures.
- **Monitoring**
 - Requires additional tools for monitoring and alerting (e.g., Cronitor) because cron itself can't alert you if jobs fail or never start.

What is DataOps?

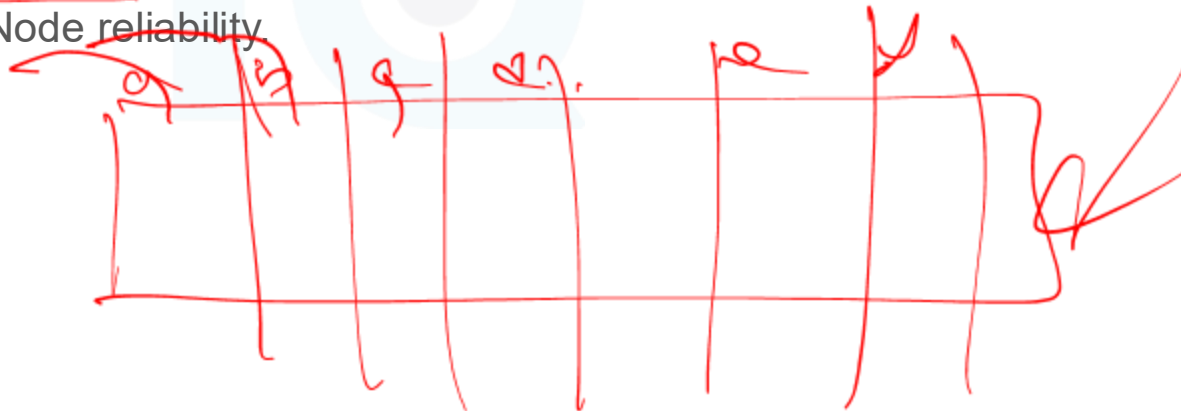
Need a Better Tool

- **High Volume Handling**
 - Capable of managing massive data volumes efficiently.
- **Scalability**
 - Easily scales out by adding more nodes to accommodate growing data.
- **Cost-Effective**
 - Utilizes commodity hardware, reducing costs compared to specialized systems.
- **Fault Tolerance**
 - Data is replicated across multiple nodes, ensuring reliability and availability even if some nodes fail.

What is DataOps?

Introduction to HDFS

- Hadoop Distributed File System, a core component of the **Hadoop** ecosystem.
- **Design**
 - Built to store large files across a distributed network of machines.
 - **Components:**
 - **NameNode**: Manages metadata and directory structure.
 - **DataNode**: Stores actual data blocks.
 - **Secondary NameNode**: Assists in checkpointing and ensures NameNode reliability.



DMP
M1

What is DataOps?

What is Hadoop?

- An open-source framework that allows for the distributed processing of large data sets across clusters of computers.
- **Core Components:**
 - **HDFS:** Storage layer.
 - **YARN:** Resource management platform.
 - **MapReduce:** Programming model for data processing.
 - **Hadoop Common:** Common utilities that support other Hadoop modules.
- **Extended Components**
 - **Hive:** Data warehousing.
 - **Pig:** Scripting language for data analysis.
 - **HBase:** NoSQL database.
 - **Spark:** In-memory data processing.

What is DataOps?

Advantages of Hadoop

- **Scalability**
 - Easily scales horizontally by adding more nodes.
- **Cost Efficiency**
 - Uses commodity hardware, making it cost-effective.
- **Flexibility**
 - Capable of processing structured, semi-structured, and unstructured data.
- **Fault Tolerance**
 - Reliable data storage and processing with automatic recovery from hardware failures.

What is DataOps?

Use Cases of Hadoop

- **Data Analytics**
 - Large-scale data analysis and business intelligence.
- **Machine Learning**
 - Training models on vast datasets.
- **Log Processing**
 - Analyzing server and application logs.
- **Ad Targeting**
 - Optimizing ad placements based on user data and clickstream analysis.

What is DataOps?

Use Cases of Hadoop

- **Data Analytics**
 - Large-scale data analysis and business intelligence.
- **Machine Learning**
 - Training models on vast datasets.
- **Log Processing**
 - Analyzing server and application logs.
- **Ad Targeting**
 - Optimizing ad placements based on user data and clickstream analysis.

What is DataOps?



What is DataOps?

- **Definition and Core Principles**

- DataOps is a set of practices, processes, and technologies that combine data engineering, data integration, data quality, and data security to deliver consistent and reliable data to data consumers (ML System in MLOps).
- **Core Principles**
 - **Collaboration:** Promotes collaboration between data engineers, data scientists, and ML Engineers (operations teams).
 - **Automation:** Emphasizes the automation of data workflows.
 - **Agility:** Focuses on the ability to rapidly adapt to changes and scale operations.

What is DataOps?

- **The Role of DataOps in Modern Data Management**
 - In modern data management, DataOps ensures that data is not only available but also reliable, accurate, and timely.
 - **Example:**
 - Consider a retail company using DataOps to manage its data. By automating data ingestion from various sources (like sales data from POS systems and customer data from CRM tools), transforming it in real-time, and ensuring its quality, the company can quickly adapt to market changes and make data-driven decisions.

Traditional Data Management vs. DataOps

- **Traditional Data Management**
 - Often siloed with manual processes.
 - Limited scalability and adaptability.
 - Delayed data delivery.
- **DataOps**
 - Integrated and automated workflows.
 - Highly scalable and flexible.
 - Real-time or near-real-time data availability.
- **Example Scenario**
 - Imagine a scenario where a healthcare organization needs to integrate and analyze patient data from multiple sources—hospital databases, wearable devices, and external labs.

Traditional Data Management vs. DataOps

- Traditional data management might involve manual processes for data collection, leading to delays and potential errors.
- With DataOps, the organization can set up automated pipelines to ingest data from all sources, transform it into a unified format, validate its quality, and store it in a scalable data lake.
- This enables real-time analytics and insights, allowing healthcare providers to make timely and informed decisions for patient care.

Traditional Data Management vs. DataOps

Limitations of Hadoop

- **Complexity in Setup and Maintenance**
 - **Initial Setup:** Setting up a Hadoop cluster involves multiple components and configurations, which can be complex and time-consuming.
 - **Ongoing Maintenance:** Requires continuous monitoring and maintenance to ensure smooth operation and performance optimization.
- **Performance Issues**
 - **Latency:** Hadoop's MapReduce model is batch-oriented, which can result in higher latency for real-time processing needs.
 - **Small Files:** HDFS is not efficient at handling a large number of small files, as it is optimized for large files. This can lead to performance bottlenecks.

Traditional Data Management vs. DataOps

Limitations of Hadoop

- **Resource Management**
 - **Inefficiency:** While YARN (Yet Another Resource Negotiator) improves resource management, it still may not be as efficient as modern resource management systems like Kubernetes for certain workloads.
 - **Resource Contention:** In a multi-tenant environment, managing resource contention and ensuring fair resource allocation can be challenging.
- **Security Concerns**
 - **Basic Security:** Hadoop's core security model is relatively basic, relying heavily on Kerberos for authentication.
 - **Data Privacy:** Ensuring data privacy and securing sensitive information require additional tools and configurations, which can add to the complexity.

Traditional Data Management vs. DataOps

Limitations of Hadoop

- **Development and Debugging**
 - **Steep Learning Curve:** Developing and debugging MapReduce jobs can be complex and requires specialized knowledge.
 - **Limited Interactive Processing:** Hadoop's batch processing nature makes interactive data processing and real-time analytics less straightforward compared to other modern data processing frameworks.
- **Hardware Dependency**
 - **Commodity Hardware:** While Hadoop is designed to run on commodity hardware, this can lead to higher failure rates and increased operational overhead in managing hardware failures.

Traditional Data Management vs. DataOps

Limitations of Hadoop

- **Ecosystem Fragmentation**
 - **Diverse Tools:** The Hadoop ecosystem includes a wide range of tools (e.g., Hive, Pig, HBase, Spark), which can lead to fragmentation and integration challenges.
 - **Version Compatibility:** Ensuring compatibility between different versions of Hadoop components can be problematic, making upgrades and migrations complex.
- **Data Management**
 - **Data Integrity:** Ensuring data integrity across a distributed file system can be challenging, requiring additional mechanisms for data validation and consistency checks.
 - **Data Movement:** Moving large datasets in and out of Hadoop can be time-consuming and resource-intensive.

Example Scenario: Scalable DataOps in Healthcare

- **Data Sources**
 - **Hospital Database:** Electronic health records (EHR).
 - **Wearable Devices:** Data from fitness trackers and health monitoring devices.
 - **External Labs:** Data from laboratory tests and external data sources.
 - **Data Ingestion: Apache NiFi/Kafka**
 - Tools used for real-time data ingestion.
 - **Data Processing: Apache Spark/Dask**
 - Distributed data processing.
 - Data being processed and transformed.
 - **Data Storage: Data Lake (e.g., S3 or HDFS):** Scalable storage solutions.
 - **Data Quality and Validation: Great Expectations/Deequ:** Tools used for validating data quality.
 - **Data Analytics and ML: Jupyter Notebook:** Data exploration and ML model development.
 - **ML Model:** Deployed machine learning models.
 - **End Users:** Doctors and healthcare professionals accessing the insights.
 - **Dashboard:** Real time analytics and dashboards.
- Handwritten notes:*
- Red arrows pointing from "Hospital Database", "Wearable Devices", and "External Labs" to "Data Ingestion".
 - Red arrow pointing from "Data Ingestion" to "Data Processing".
 - Red arrow pointing from "Data Processing" to "Data Storage".
 - Red arrow pointing from "Data Storage" to "Data Quality and Validation".
 - Red arrow pointing from "Data Quality and Validation" to "Data Analytics and ML".
 - Red arrow pointing from "Data Analytics and ML" to "ML Model".
 - Red arrow pointing from "ML Model" to "End Users".
 - Red arrow pointing from "End Users" to "Dashboard".
 - Red arrow pointing from "Dashboard" back to "Data Sources".
 - Red box around "Data Processing: Apache Spark/Dask" with note: "O.K. [if GPU is preferred]".
 - Red box around "Data Analytics and ML: Jupyter Notebook" with note: "Scik learn / JAX / keras / TF".

Real-World Example: Netflix's Scalable DataOps Pipeline

- **Overview of Netflix's Data Infrastructure and ML Use Cases**
 - Netflix handles massive volumes of data generated from users' viewing habits, content interactions, and system operations.
- **How Netflix Leverages DataOps Principles**
 - **Data Ingestion:** Uses Apache Kafka for real-time data ingestion.
 - **Data Processing:** Utilizes Apache Spark for distributed data processing.
 - **Data Storage:** Stores data in Amazon S3 for scalable data storage.
 - **Exploration & Orchestration:** Employs Jupyter Notebooks and Airflow for data exploration and pipeline orchestration.
- **Benefits Realized by Netflix:**
 - Improved data quality and consistency.
 - Faster experimentation and model deployment.
 - Increased collaboration between data teams.
- **Key Takeaway:**
 - Through scalable DataOps practices, Netflix can efficiently manage its data workflows, ensuring high-quality data is readily available for machine learning models and analytical insights.



“Building and Managing Scalable DataOps Pipelines for ML

Building and Managing Scalable DataOps Pipelines for ML

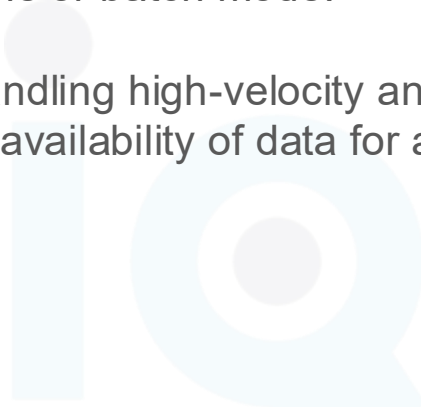
Importance of Scalable DataOps in ML

- **DataOps Overview**
 - DataOps is a set of practices and technologies designed to improve the quality and speed of data analytics.
 - Focuses on collaboration, automation, and monitoring across the data lifecycle.
- **Importance in ML**
 - Ensures efficient handling of large datasets.
 - Speeds up the ML model development and deployment process.
 - Facilitates continuous integration and delivery of data and models.
- **Example**
 - A retail company uses DataOps to streamline data processing and model deployment, reducing the time from data ingestion to actionable insights from weeks to hours.

Scalable DataOps Pipeline Components

Scalable Data Ingestion

- The process of importing, processing, and storing large volumes of data from various sources in real-time or batch mode.
 - **Importance**
 - Essential for handling high-velocity and high-volume data streams.
 - Ensures timely availability of data for analysis and machine learning.

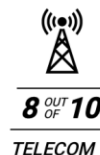
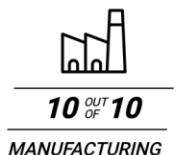


Scalable DataOps Pipeline Components

Scalable Data Ingestion

- **Apache Kafka**

- An open-source platform
- Distributed streaming platform for building real-time data pi
- Handles high-throughput, low-latency data ingestion.
- **Key Features:**
 - **Scalability:** Easily scales horizontally by adding more brokers.
 - **Durability:** Ensures data persistence through distributed logs.
 - **Fault Tolerance:** Replicates data across multiple nodes.
 - Example: A financial services firm can use Kafka to ingest and process real-time transaction data for fraud detection.



<https://kafka.apache.org>

Scalable DataOps Pipeline Components

Scalable Data Ingestion


- **Apache NiFi**
 - Data integration tool for automating the flow of data between systems.
 - Provides a web-based interface for designing data flows.
 - An easy to use, powerful, and reliable system to process and distribute data
 - **Key Features:**
 - **Scalability:** Scales vertically and horizontally with clustering.
 - **Data Provenance:** Tracks data flow history for auditing.
 - **Ease of Use:** Drag-and-drop interface for building complex data flows.
 - **Example:** A healthcare provider can use NiFi to ingest data from multiple sources (e.g., EHR systems, IoT devices) and route it to a centralized data lake



<https://nifi.apache.org>

Scalable DataOps Pipeline Components

Scalable Data Ingestion

-  **Amazon Kinesis**
 - Managed service for real-time data streaming on AWS.
 - Supports ingestion, processing, and analysis of streaming data.
 - **Key Features:**
 - **Scalability:** Automatically scales to match data throughput.
 - **Reliability:** Guarantees data delivery with multiple replication.
 - **Integration:** Seamlessly integrates with other AWS services.
 - **Example:** An e-commerce company uses Kinesis to capture and analyze clickstream data in real-time to personalize user experiences.

Scalable DataOps Pipeline Components

Scalable Data Ingestion



Scalable DataOps Pipeline Components

Comparing Ingestion Tools

- **Apache Kafka**
 - Best for high-throughput, low-latency data streaming.
 - Suitable for real-time analytics and event sourcing.
- **Apache NiFi**
 - Ideal for complex data flows and data provenance.
 - User-friendly interface for designing data pipelines.
- **Amazon Kinesis**
 - Easy to use and integrates well with AWS ecosystem.
 - Automatically scales based on data throughput.

Scalable DataOps Pipeline Components

Comparing Ingestion Tools

Feature	Apache Kafka	Apache NiFi	Amazon Kinesis
Purpose	Real-time data streaming and pipeline building	Data flow automation and integration	Real-time data streaming and processing
Scalability	Horizontally scalable by adding more brokers	Scales vertically and horizontally with clustering	Automatically scales based on data throughput
Latency	Low latency for high-throughput data ingestion	Low to moderate latency depending on configuration	Low latency with high throughput
Data Persistence	Persistent storage with distributed logs	Persistent storage with data provenance	Data retention configurable up to 7 days by default
Fault Tolerance	Replicates data across multiple nodes for redundancy	Built-in fault tolerance through clustering	Built-in fault tolerance with multiple replicas
Ease of Use	Requires configuration and management	User-friendly drag-and-drop interface for designing	Managed service with easy setup and

Vendor lock

Scalable DataOps Pipeline Components

Comparing Ingestion Tools

Feature	Apache Kafka	Apache NiFi	Amazon Kinesis
Integration	Integrates with various stream processing systems	Integrates with a wide range of data sources and sinks	Seamless integration with AWS services (e.g., Lambda, S3)
Data Provenance	Limited built-in support	Comprehensive data provenance tracking	Limited built-in support
Security	Supports SSL/TLS encryption and authentication	Supports SSL/TLS, user authentication, and role-based access control	Supports encryption at rest and in transit, IAM policies
Use Cases	Real-time analytics, event sourcing, log aggregation	Complex data flows, ETL processes, data provenance	Real-time analytics, log and event stream processing
Example	A financial services firm using Kafka for real-time fraud detection	A healthcare provider using NiFi for ingesting data from EHR systems and IoT devices	An e-commerce company using Kinesis for real-time clickstream

Scalable DataOps Pipeline Components

Weaknesses of Data Ingestion Tools

- **Apache Kafka**
 - **Complexity:** Kafka has a steep learning curve and requires significant setup and configuration efforts, especially for beginners.
 - **Lack of built-in data transformation:** Kafka does not provide built-in capabilities for complex data transformations, requiring additional tools or custom code.
 - **Limited data provenance support:** Kafka has limited built-in support for data provenance tracking, which may be a concern for applications with strict auditing and compliance requirements.

Scalable DataOps Pipeline Components

Weaknesses of Data Ingestion Tools

- **Apache NiFi**
 - **Scalability limitations:** While NiFi supports clustering for scalability, it may face challenges in handling extremely high data volumes compared to tools like Kafka.
 - **Processing overhead:** NiFi's extensive data provenance tracking and flow-based design can introduce processing overhead, potentially impacting performance in high-throughput scenarios.
 - **Learning curve:** NiFi's drag-and-drop interface and extensive features may require some learning and familiarity for effective use.
- **Amazon Kinesis**
 - **Vendor lock-in:** As a managed service by AWS, Kinesis may lead to vendor lock-in, making it harder to switch to other platforms if needed.
 - **Cost considerations:** Kinesis pricing is based on factors like data volume, shard hours, and data retention, which can become costly for applications with high data throughput and long retention periods.
 - **Limited customization options:** Being a managed service, Kinesis offers less flexibility and customization compared to open source alternatives like

Scalable DataOps Pipeline Components

Choosing the Right Tool

- **Apache Kafka**
 - Best for high-throughput, real-time data streaming and event sourcing.
 - Suitable for applications requiring persistent, distributed logs and low latency.
- **Apache NiFi**
 - Ideal for complex data flows, ETL processes, and data provenance tracking.
 - Suitable for applications needing a user-friendly interface for designing and monitoring data flows.
- **Amazon Kinesis**
 - Best for real-time data streaming and seamless integration with AWS services.
 - Suitable for applications requiring automatic scaling and real-time analytics capabilities.

“

Data Processing and Transformation with Apache Spark and Dask

Scalable Data Processing and Transformation

Overview

- Importance of scalable data processing and transformation.
- **Key tools:** Apache Spark and Dask.
- **Goals:** Handle large datasets efficiently, enable real-time processing, and support complex transformations.



Scalable Data Processing and Transformation

Apache Spark: An Overview

- Apache Spark is an open-source unified analytics engine for large-scale data processing.
- **Key Features**
 - In-memory computation.
 - Distributed data processing.
 - High-level APIs in Java, Scala, Python, and R.
 - Libraries for SQL, streaming, machine learning (MLlib), and graph processing (GraphX).
- **Use Cases**
 - Batch processing.
 - Real-time stream processing.
 - Interactive data analytics.
 - Machine learning workflows.

Scalable Data Processing and Transformation

Apache Spark Architecture

- **Components:**
 - **Driver Program:** Manages the execution of the application.
 - **Cluster Manager:** Allocates resources to applications.
 - **Workers:** Execute tasks on the clusters.
 - **Executors:** Run on worker nodes to execute tasks and store data.
- **Data Structures:**
 - **RDD (Resilient Distributed Dataset):** Immutable distributed collections of objects.
 - **DataFrame:** Distributed collection of data organized into named columns.
 - **Dataset:** Extension of DataFrame with type-safe operations.

Scalable Data Processing and Transformation

Example: Processing Data with Apache Spark

- **Scenario:** Processing a large log file to extract error messages and compute statistics.

- Read

```
from pyspark.sql import SparkSession
spark = SparkSession.builder.appName("LogProcessor").getOrCreate()
logs = spark.read.text("path/to/logfile")
```

- Filter

```
errors = logs.filter(logs.value.contains("ERROR"))
```

- Count

```
error_counts = errors.groupBy("value").count()
error_counts.show()
```

Check: <https://spark.apache.org/examples.html>

Scalable Data Processing and Transformation

Dask Overview

- Dask is a flexible parallel computing library for analytic computing in Python.
- **Key Features**
 - Parallel collections (arrays, dataframes, bags).
 - Dynamic task scheduling.
 - Scales from a single machine to a cluster.
 - Integrates with existing Python ecosystems (NumPy, Pandas, Scikit-Learn).
- Use Cases
 - Large-scale data processing.
 - Real-time analytics.
 - Parallelizing custom workflows.

Scalable Data Processing and Transformation

Dask Architecture

- Components
 - **Scheduler**: Coordinates the execution of tasks.
 - **Workers**: Execute tasks and store data.
 - **Client**: Interface for users to interact with the Dask cluster.
- Data Structures
 - Dask Array: Parallel NumPy array.
 - Dask DataFrame: Parallel Pandas DataFrame.
 - Dask Bag: Parallel list for processing unstructured data.
- Task Graphs
 - Visual representation of the computation.
 - Optimizes and schedules tasks dynamically.

Scalable Data Processing and Transformation

Example: Processing Data with Dask

- **Scenario:** Processing a large CSV file to compute aggregate statistics.

- Read the Data:

```
import dask.dataframe as dd
df = dd.read_csv("path/to/largefile.csv")
```

- Filter the data

```
filtered_df = df[df['column_name'] > threshold]
```

- Compute Aggregates

```
result = filtered_df.groupby('another_column').mean().compute()
print(result)
```

Scalable Data Processing and Transformation

Comparing Apache Spark and Dask

- **Apache Spark**
 - **Strengths:** In-memory processing, extensive libraries, robust community support.
 - **Use Cases:** Batch processing, real-time stream processing, ML workflows.
- **Dask**
 - **Strengths:** Integration with Python ecosystem, flexibility, dynamic task scheduling.
 - **Use Cases:** Custom workflows, real-time analytics, interactive data analysis.
- **Choosing the Right Tool:**
 - Consider the scale of data, type of workflows, and existing infrastructure.
 - Example: Spark for large-scale batch processing, Dask for flexible and interactive analytics.

Scalable Data Processing and Transformation

Best Practices for Scalable Data Processing

- **Design Principles**
 - Modular and reusable code.
 - Efficient data partitioning.
 - Lazy evaluation for optimal resource use.
- **Performance Optimization**
 - Use in-memory processing when possible.
 - Optimize data shuffling and partitioning.
 - Monitor and profile performance regularly.
- **Scalability Considerations**
 - **Horizontal scaling:** Add more nodes to the cluster.
 - **Vertical scaling:** Enhance the capacity of existing nodes.
 - Use cloud-native solutions for elasticity.

Scalable Data Processing and Transformation

Summary

- Importance of scalable data processing.
- Key tools: Apache Spark and Dask.
- Practical examples and best practices.
- Next Steps:
 - Hands-on practice with Spark and Dask.
 - Explore further resources and documentation.

“Scalable Data Storage

Scalable Data Storage

Introduction

- **Importance of Scalable Data Storage**
 - Handles vast amounts of data efficiently.
 - Supports high-throughput data ingestion and querying.
 - Ensures data durability and availability.
- **Key Concepts:**
 - **Data Lakes:** Centralized repositories for storing raw and processed data.
 - **Data Lakehouses:** Combines the best features of data lakes and data warehouses.

Scalable Data Storage

Data Lakes

- A data lake is a centralized repository that allows you to store all your structured and unstructured data at any scale.
- **Characteristics**
 - Stores raw data in its native format.
 - Supports various data types (structured, semi-structured, unstructured).
 - High scalability and cost-effective storage.
 - Examples
 - Amazon S3: Scalable object storage service.
 - Features: High availability, durability, and security.
 - Example Use Case: Storing raw log files for big data analytics.
- Hadoop Distributed File System (HDFS)
 - Designed for large-scale data processing.
 - Features: Fault-tolerant, scalable, and distributed.
 - Example Use Case: Storing large datasets for Hadoop-based processing.

Scalable Data Storage

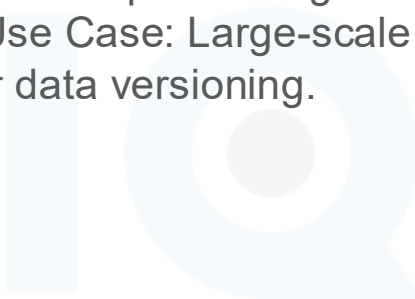
Data Lakehouses

- A data lakehouse is an architecture that combines the storage flexibility of data lakes with the data management capabilities of data warehouses.
- **Characteristics**
 - Unified storage layer for both raw and processed data.
 - Supports ACID (Atomicity, Consistency, Isolation, and Durability) transactions and schema enforcement.
 - Enables efficient data processing and analytics.
 - Examples:
 - Delta Lake: Open-source storage layer that brings ACID transactions to Apache Spark.
 - Features: ACID transactions, scalable metadata handling, and unified batch/streaming data processing.

Scalable Data Storage

Data Lakehouses

- Apache Hudi
 - Manages storage for large analytical datasets.
 - Features: Support for ACID transactions, efficient upserts, and incremental data processing.
 - Example Use Case: Large-scale data ingestion and ETL with support for data versioning.



Scalable Data Storage

Amazon S3 for Data Lakes

- Scalable object storage service by AWS.
- Key Features:
 - Scalability: Automatically scales to handle growing data volumes.
 - Durability: 99.999999999% (11 9's) durability.
 - Security: Supports encryption and access control policies.
 - Example:
 - Use Case: Storing raw log data for an e-commerce platform.
 - Scenario: Logs from web servers, application servers, and databases are ingested into S3 for later analysis.
 - Benefits: Cost-effective storage, easy integration with AWS analytics services (e.g., Athena, Redshift).

Scalable Data Storage

HDFS for Data Lakes

- Distributed file system designed for large-scale data processing.
- **Key Features**
 - **Scalability:** Can scale out by adding more nodes.
 - **Fault Tolerance:** Data replication across multiple nodes.
 - **High Throughput:** Optimized for large sequential reads and writes.
 - Example:
 - Use Case: Storing large datasets for a Hadoop-based analytics platform.
 - Scenario: Data from various sources (e.g., IoT devices, social media) is stored in HDFS for batch processing using Hadoop.
 - Benefits: High throughput, fault tolerance, integration with Hadoop ecosystem (e.g., MapReduce, Hive).

Scalable Data Storage

Delta Lake for Data Lakehouses

- Open-source storage layer that adds ACID transactions to Apache Spark.
- Key Features
 - ACID Transactions: Ensures data reliability and consistency.
 - Schema Enforcement: Supports schema evolution and enforcement.
 - Unified Batch/Streaming: Handles both batch and streaming data.
 - Example:
 - Use Case: Real-time analytics on IoT data.
 - Scenario: IoT sensor data is ingested in real-time into Delta Lake, enabling up-to-date analytics using Spark.
 - Benefits: Real-time data processing, reliable data transactions, simplified data pipeline.

Scalable Data Storage

Apache Hudi for Data Lakehouses

- Manages large analytical datasets with support for incremental data processing.
- Key Features
 - ACID Transactions: Ensures data consistency.
 - Efficient Upserts: Optimized for updating and deleting records.
 - Incremental Processing: Supports efficient ETL and data ingestion.
 - Example:
 - Use Case: Incremental data ingestion for a financial analytics platform.
 - Scenario: Financial transaction data is ingested incrementally into Apache Hudi, enabling efficient ETL and data querying.
 - Benefits: Efficient data ingestion, support for data versioning, reduced processing latency.

Scalable Data Storage

Summary

- Data lakes (S3, HDFS) are essential for scalable storage of raw and processed data.
- Data lakehouses (Delta Lake, Apache Hudi) combine the best features of data lakes and data warehouses, supporting advanced analytics and data management.
- Importance
 - Scalable data storage solutions are crucial for handling the growing volume and variety of data in modern AI and ML applications.
- Next Steps
 - Explore and implement scalable data storage solutions in your projects.
 - Stay updated with the latest advancements in data storage technologies.

“Data Validation at Scale

Data Validation at Scale

Introduction

- Data validation is the process of ensuring that data is accurate, clean, and useful.
- **Importance**
 - Ensures data quality and integrity.
 - Prevents errors in downstream ML models.
 - Facilitates compliance with data governance policies.

Data Validation at Scale

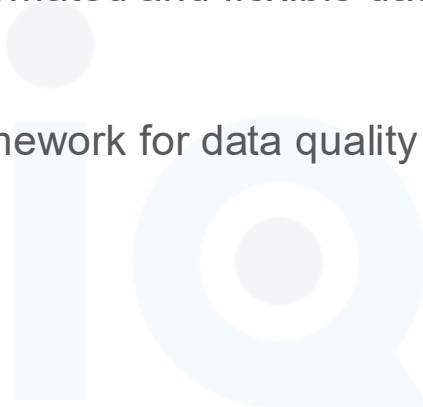
Challenges in Data Validation at Scale

- Volume
 - Handling large datasets efficiently.
- Variety
 - Dealing with diverse data types and sources.
- Velocity
 - Validating streaming data in real-time.
- Veracity
 - Ensuring data reliability and accuracy.

Data Validation at Scale

Tools for Data Validation at Scale

- Great Expectations
 - A robust tool for automated and flexible data validation.
- Apache Griffin
 - An open-source framework for data quality management and validation.



Data Validation at Scale

Using Great Expectations - Example

- **Scenario:** Validating a dataset of customer transactions.
- Initialize a Data Context

```
import great_expectations as ge
context = ge.data_context.DataContext()
```

- Create Expectations

```
# Define expectations for the dataset
expectations_suite = context.create_expectation_suite("transactions_suite",
overwrite_existing=True)
batch = context.get_batch(batch_kwarg, expectations_suite)
batch.expect_column_values_to_be_of_type("transaction_id", "int")
batch.expect_column_values_to_not_be_null("customer_id")
batch.expect_column_values_to_match_regex("email", r"^[^@]+@[^@]+\.[^@]+$")
```

- Validate Data

```
results = context.run_validation_operator("action_list_operator", assets_to_validate=[batch])
```

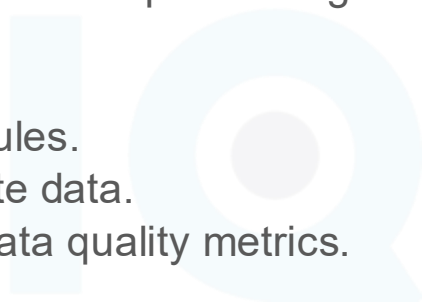
- View Validation Results

```
context.build_data_docs()
context.open_data_docs()
```


Data Validation at Scale

Apache Griffin Overview

- Key Features
 - Supports batch and streaming data validation.
 - Provides a central dashboard for monitoring data quality.
 - Integrates with various data processing frameworks (Spark, Flink).
- How It Works
 - Define data quality rules.
 - Apply rules to validate data.
 - Monitor and report data quality metrics.



Data Validation at Scale

Using Apache Griffin - Example

- **Scenario:** Ensuring data quality in a batch processing pipeline.
- Define Data Quality Rule

```
{
  "name": "CustomerDataQuality",
  "description": "Data quality rules for customer data",
  "rules": [
    {
      "name": "NonNullCustomerID",
      "dqType": "NON_NULL",
      "source": "customer_data",
      "target": "customer_id",
      "expression": "customer_id IS NOT NULL"
    },
    {
      "name": "ValidEmailFormat",
      "dqType": "REGEX_MATCH",
      "source": "customer_data",
      "target": "email",
      "expression": "[^@]+@[^@]+\.\.[^@]+"
    }
  ]
}
```

Data Validation at Scale

Using Apache Griffin - Example

- Configure Data Sources
 - Define data sources in the Griffin configuration file to connect to your data storage.
- Run Data Validation
 - Use Griffin CLI or integrate with your data processing job to run the validation.
- Monitor Results
 - Access the Griffin dashboard to view validation results and metrics.

Data Validation at Scale

Comparing Great Expectations and Apache Griffin

- Great Expectations
 - Best for flexible, detailed data validation.
 - Strong integration with various data sources.
 - Excellent documentation and community support.
- Apache Griffin
 - Ideal for large-scale, enterprise data quality management.
 - Supports both batch and streaming data validation.
 - Comprehensive monitoring and reporting capabilities.

Data Validation at Scale

Best Practices for Scalable Data Validation

- **Automate Validation**
 - Integrate validation into CI/CD pipelines.
- **Monitor Continuously**
 - Use monitoring tools to keep track of data quality in real-time.
- **Document Expectations**
 - Maintain thorough documentation of data quality rules and expectations.
- **Iterate and Improve**
 - Regularly review and update validation rules based on feedback and new requirements.

Data Validation at Scale

Summary

- Importance of data validation at scale.
- Tools like Great Expectations and Apache Griffin can simplify and automate the process.



“Designing Scalable DataOps Pipelines

Designing Scalable DataOps Pipelines

Principles of Scalable Pipeline Architecture

- **Key principles**
 - decoupling, modularity, and fault tolerance.

Decoupling

- Decoupling refers to designing systems where different components operate independently of each other.
- **Benefits:**
 - **Scalability:** Independent scaling of components based on demand.
 - **Flexibility:** Easier to replace or upgrade individual components without affecting others.
 - **Resilience:** Failures in one component do not impact the entire system

Designing Scalable DataOps Pipelines

Decoupling

- **Implementation Strategies**
 - **Microservices Architecture**
 - Break down monolithic applications into smaller, independent services.
 - **Example:** Separating data ingestion, processing, and storage into distinct microservices.
 - **Message Queues**
 - Use message queues (e.g., Apache Kafka, RabbitMQ) to decouple data producers and consumers.
 - Example: Data ingestion service sends data to Kafka, and multiple processing services consume data from Kafka independently.

Designing Scalable DataOps Pipelines

Modularity

- Modularity involves designing systems as a collection of distinct modules, each responsible for a specific function.
- **Benefits**
 - **Maintainability:** Easier to manage, test, and debug individual modules.
 - **Reusability:** Modules can be reused across different projects or pipelines.
 - **Scalability:** Modules can be independently scaled to handle varying loads.
- **Implementation Strategies**
 - **Containerization**
 - Use containers (e.g., Docker) to package and deploy modules independently.
 - **Example:** Containerize ETL jobs, model training scripts, and serving endpoints.
 - **Function as a Service (FaaS)**
 - Use serverless functions (e.g., AWS Lambda, Azure Functions) to implement modular and scalable components.
 - **Example:** Serverless functions for data transformation tasks that can scale based on incoming data volume.

Designing Scalable DataOps Pipelines

Fault Tolerance

- Fault tolerance is the ability of a system to continue operating properly in the event of a failure of some of its components.
- **Benefits:**
 - **Reliability:** Ensures continuous operation and availability of services.
 - **Resilience:** Minimizes the impact of failures on the overall system.
 - **User Experience:** Provides a seamless experience despite underlying issues.
- **Implementation Strategies**
 - **Redundancy**
 - Use redundant instances of critical components to ensure availability.
 - **Example:** Deploy multiple instances of data ingestion services across different regions.
 - **Failover Mechanisms**
 - Implement automatic failover to backup systems in case of component failure.
 - **Example:** Use load balancers to redirect traffic to healthy instances during failures.
 - **Error Handling and Retries:**
 - Implement robust error handling and retry mechanisms.

Example: Retries for failed data processing jobs, with exponential backoff and

Designing Scalable DataOps Pipelines

Scalable DataOps Pipeline Architecture

- A sample architecture demonstrating the principles of decoupling, modularity, and fault tolerance.
- **Components:**
 - **Data Ingestion**
 - Decoupled using Apache Kafka as the message queue.
 - Multiple ingestion services running in containers.
 - **Data Processing**
 - Modular ETL jobs implemented as serverless functions (AWS Lambda).
 - Parallel processing using Apache Spark.
 - **Data Storage**
 - Scalable storage using Amazon S3.
 - Metadata management with Apache Atlas.

Designing Scalable DataOps Pipelines

Scalable DataOps Pipeline Architecture

- **Monitoring and Alerting**
 - Prometheus for monitoring.
 - Grafana for visualization and alerting.
- **Fault Tolerance**
 - Redundant Kafka brokers.
 - Automatic failover for ETL functions using AWS Lambda.
 - Error handling and retries with alerting mechanisms in place.

Designing Scalable DataOps Pipelines

Case Study: Real-World Implementation

- **Background**
 - Company-X faced challenges with scaling their data processing pipeline.
- **Solution**
 - Implemented decoupling with Apache Kafka for data ingestion.
 - Used Docker for containerizing modular processing jobs.
 - Ensured fault tolerance with redundant components and automatic failover.
- **Results**
 - Enhanced scalability, allowing the pipeline to handle 10x the data volume.
 - Improved system reliability and reduced downtime by 90%.
 - Increased flexibility, enabling rapid deployment of new processing modules.

Designing Scalable DataOps Pipelines

Summary

- **Recap of Principles**
 - **Decoupling**: Independent scaling, flexibility, and resilience.
 - **Modularity**: Maintainability, reusability, and independent scaling.
 - **Fault Tolerance**: Reliability, resilience, and seamless user experience.
- **Importance of Scalable Architecture**
 - Ensures systems can grow with data and computational demands.
 - Provides a foundation for robust, efficient, and maintainable pipelines.
- **Next Steps**
 - Explore tools and frameworks for implementing these principles.
 - Apply these principles to your DataOps pipelines for improved scalability and reliability.

“Scalable ML Integration

Scalable ML Integration

Introduction

- Understand the principles and tools for scalable ML model training, deployment, and monitoring.
- Explore practical examples and best practices.

Why Distributed Training?

- Handle large datasets that don't fit into memory.
- Speed up training by parallelizing computations.
- Improve model performance by leveraging multiple GPUs/CPU.

Scalable ML Integration

Tools Overview

- **Horovod**
 - Developed by Uber for distributed deep learning.
 - Compatible with TensorFlow, Keras, PyTorch, and Apache MXNet.
 - Uses MPI or Gloo for inter-process communication.
- **Dask-ML**
 - Scalable machine learning with Dask.
 - Integrates with Scikit-Learn for parallel processing.
 - Handles large-scale data processing tasks.
- **Spark MLlib**
 - Part of Apache Spark for scalable machine learning on big data.
 - Supports various ML algorithms like classification, regression, clustering, etc.
 - Optimized for distributed computing environments.

Scalable ML Integration

Horovod

```
import horovod.tensorflow as hvd

# Initialize Horovod
hvd.init()

# Pin GPU to be used by each process
config = tf.ConfigProto()
config.gpu_options.visible_device_list = str(hvd.local_rank())

# Scale the training
opt = hvd.DistributedOptimizer(tf.train.AdamOptimizer(learning_rate=0.001))

# Broadcast initial variables
hvd.broadcast_variables(model.variables, root_rank=0)

# Train the model
model.fit(dataset, epochs=10)
```

Scalable ML Integration

Dask-ML

```
import dask.dataframe as dd
from dask_ml.model_selection import train_test_split
from dask_ml.linear_model import LogisticRegression

# Load data into a Dask DataFrame
df = dd.read_csv('data.csv')

# Split data
X = df.drop('target', axis=1)
y = df['target']
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)

# Train the model
model = LogisticRegression().fit(X_train, y_train)
```

Scalable ML Integration

Spark MLlib

```
from pyspark.sql import SparkSession
from pyspark.ml.classification import LogisticRegression

# Initialize Spark session
spark = SparkSession.builder.appName('MLlibExample').getOrCreate()

# Load data
data = spark.read.csv('data.csv', header=True, inferSchema=True)

# Initialize and train the model
lr = LogisticRegression(featuresCol='features', labelCol='label')
model = lr.fit(data)
```

Scalable ML Integration

Scalable Model Deployment

- **Kubernetes**
 - Container orchestration platform.
 - Manages deployment, scaling, and operations of application containers.
 - Suitable for deploying ML models as microservices.
- **Kserve** (<https://github.com/kserve/kserve>)
 - KFServing has been rebranded to KServe since v0.7.
 - Provides Kubernetes-native model serving capabilities.
 - Supports multiple ML frameworks (TensorFlow, PyTorch, XGBoost).
 - Enables easy scaling and management of model inference.
- **Mlflow**
 - Open-source platform for managing the ML lifecycle.
 - Features: tracking experiments, packaging code into reproducible runs, and sharing and deploying models.
 - Supports deployment on various platforms (local, cloud, Kubernetes).

Scalable ML Integration

ML Monitoring and Retraining at Scale

- **Monitoring Data Drift and Model Performance**
 - **Tools:** Evidently AI, Prometheus, Grafana
 - **Key metrics:** accuracy, precision, recall, F1 score, latency
- **Automating Model Retraining**
 - Trigger retraining on data drift detection.
 - Use CI/CD pipelines to automate retraining and deployment.
 - Tools: Jenkins, GitLab CI/CD, Apache Airflow

Scalable ML Integration

Example: Automating Model Retraining

- Setup
 - Integrate monitoring tools (e.g., Prometheus) with your ML pipeline.
 - Set up CI/CD pipeline (e.g., Jenkins) for retraining.
- CI/CD Pipeline Configuration
 - Define stages: data ingestion, model training, validation, deployment.

```
pipeline {
  agent any
  stages {
    stage('Data Ingestion') {
      steps {
        // Script for data ingestion
      }
    }
    stage('Model Training') {
      steps {
        // Script for training the model
      }
    }
    stage('Model Validation') {
      steps {
        // Script for validating the model
      }
    }
    stage('Deployment') {
      steps {
        // Script for deploying the model
      }
    }
  }
}
```


Scalable DataOps for Machine Learning

Summary

- **Overview:** Principles, best practices, and tools for scalable DataOps pipelines in ML.
- **Importance:** Handling large data volumes, real-time processing, resource optimization, system reliability.
- **Data Ingestion:** Tools - Apache Kafka, Apache NiFi, Amazon Kinesis.
- **Data Processing:** ETL vs. ELT, Tools - Apache Spark, Dask.
- **Data Storage:** Solutions - HDFS, S3, Google Cloud Storage, Delta Lake.
- **Data Quality:** Tools - Great Expectations, Apache Griffin.
- **Architecture:** Decoupling, modularity, fault tolerance.
- **Orchestration:** Tools - Apache Airflow, Prefect, Apache NiFi.
- **Monitoring:** Tools - Prometheus, Grafana, ELK stack.
- **Case Study:** Real-world scalable DataOps pipeline example.

Scalable DataOps for Machine Learning

Summary

- **Distributed Training:** Tools - Horovod, Dask-ML, Spark MLlib.
- **Model Deployment:** Kubernetes, KFServing, MLflow.
- **Monitoring/Retraining:** Tools - Evidently AI, Prometheus, Grafana, CI/CD with Jenkins.



AI System @ Scale

