1. Platform Architecture Diagram and Components

- **Components**: AWS S3, Apache Iceberg, Apache Airflow, Apache Spark, Starburst, Ranger, Vector DB
- **Architecture Diagram**: Showcase how data flows from the source (SOR) to ingestion, processing, quality checks, and finally, consumption, across the three layers (Staging, Standardization, and Consumption).
- **Key Areas**:
  - Ingestion Service (Batch/File-Based/Vector DB)
  - Data Quality and Validation Service
  - Data Entitlement, Encryption, and Masking
  - Orchestration and Monitoring

2. Design the Schema, Bucket, Folder Structure for Staging, Standardization, and Consumption Layers

- **Staging Layer**:
  - S3 folder structure (e.g., `/staging/{source}/{table}/{year}/{month}/{day}`)
  - Schema design for raw data
- **Standardization Layer**:
  - Converting raw data into standardized formats (Parquet, ORC) using Iceberg
  - Versioning and partitioning rules
- **Consumption Layer**:
  - Simplified access for end-users using Starburst SQL access
  - Aggregations, views, and derived datasets

3. Design Iceberg Table Formats, Compression, Partitioning

- **Table Formats**: Iceberg format, supporting schema evolution, time travel, and partitioning.
- **Partitioning Strategy**: Based on business dimensions like date, geography, or product.
- **Compression**: Snappy, ZSTD for optimized storage and query performance.

4. Design Common Data Ingestion Service (Batch, File-Based, Vector DB)

- **Ingestion Framework**:
  - Apache Spark jobs for file-based ingestion.
  - Real-time CDC using Airflow.
  - Configuration-driven ingestion service for S3 and Vector DB.
  - Error handling and retry mechanisms.

5. Design Data Quality and Validation Service

- **Framework**:
  - Airflow DAGs for running data validation checks.
  - Pre-built quality checks (row count, null checks, business rule validation).
  - Customizable rules and notifications for failures.

6. Design Data Transformation & Processing Service (Batch, Micro-Batch, Streaming)

- **Batch Processing**: Apache Spark jobs scheduled via Airflow, performing ETL jobs.

- **Micro-Batch Streaming**: Real-time ingestion using Spark Structured Streaming.
- **Transformation Rules**: Standardizing, cleansing, and aggregating data.

7. Design Data Testing Automation Service

- **Test Automation**:
  - Automating test data validation using Airflow and custom scripts.
  - Regression testing for schema changes.
  - Automated validation reports.

8. Design Orchestration Service

- **Airflow DAGs**:
  - Manage workflows (e.g., ingestion, processing, quality checks).
  - Dependencies between services (e.g., Iceberg table updates after ingestion).

9. Design Data Encryption at Rest, Transit, Masking Service

- **Encryption**:
  - S3 SSE (Server-Side Encryption) for data at rest.
  - TLS encryption for data in transit.
- **Data Masking**:
  - Role-based access using Apache Ranger to mask sensitive fields.

10. Design Data Entitlement Service

- **Entitlement and Authorization**:
  - Apache Ranger policies for access control.
  - Integration with Active Directory for user management.

11. Design Data Catalog Service

- **Cataloging**:
  - Centralized data catalog using Iceberg and Starburst metadata.
  - Cataloging of schemas, tables, and lineage tracking.

12. Design Data Consumption/Distribution Service

- **Data Distribution**:
  - Access through Starburst's distributed SQL engine.
  - Ad-hoc query support, dashboard integrations.

13. Design Monitoring, Logging, Alert Service

- **Monitoring Tools**:
  - Integration with Prometheus, Grafana for real-time monitoring.
  - Logging mechanisms in Spark and Airflow, centralized in ELK (Elasticsearch, Logstash, Kibana).
  - Alerting for job failures, resource usage spikes.

14. Design Data Purging, Compaction, and Archival/Retrieval Service

- **Purging and Archiving**:
  - Periodic purging of obsolete data in S3 (lifecycle policies).
  - Data compaction using Iceberg for performance improvements.
  - Archival policies for long-term storage.

Next Steps:

- Create detailed flow diagrams for each component.
- Define SLAs and resource requirements for the Spark jobs.
- Integrate service-level monitoring and alerting for early detection of issues.

This document will provide a comprehensive guide for implementing the modern data platform based on the technologies listed (S3, Iceberg, Spark, etc.).

1. Overview of Data Quality and Validation Service

The Data Quality (DQ) and Validation Service ensures the integrity, accuracy, and completeness of data before it is processed or moved to the next stage in the data pipeline. This service will be automated using Airflow and integrated with Spark for performance scalability.

Key Objectives:

- **Data Integrity**: Ensure data complies with business rules and meets quality standards.
- **Automated Validation**: Execute quality checks automatically at different stages of the data pipeline.
- **Error Reporting**: Provide detailed reports on data quality failures with error logs for remediation.
- **Scalability**: Handle large data volumes and support batch, micro-batch, and streaming data validation.
- **Real-time Monitoring**: Ensure timely identification of data quality issues.

2. Functional Requirements

2.1. Pre-Ingestion Quality Checks

- Schema validation (data type, length, and mandatory fields).
- Duplication checks (check for duplicate records).
- Reference data validation (checking for valid lookup values).
- Null value checks for non-nullable fields.

2.2. Post-Ingestion Quality Checks

- Data format validation (ensure correct file format - e.g., Parquet/CSV).
- Consistency checks (e.g., foreign key relationships, range values).
- Business rule validation (e.g., price must be positive, date should be in a valid range).
- Record count validation between ingestion source and data platform.

2.3. Data Profiling

- Generate statistics such as max, min, average, and record count.
- Provide insight into distributions, cardinality, and anomalies.

2.4. Error Logging & Notification

- Log errors to a centralized system (e.g., AWS CloudWatch or ELK Stack).
- Generate detailed error reports with error type, failing record details, and recommended remediation.
- Send alerts to defined stakeholders when data quality rules fail (using Slack, email, etc.).

2.5. Retrying Failed Data Loads

- Automatic retry mechanism in case of transient issues.
- Manual retry capability after data issues are fixed.

3. Technical Design

3.1. Architecture

- **Apache Airflow**: Orchestrating the quality checks in a DAG (Directed Acyclic Graph).
  - Each DAG will consist of different steps (e.g., schema validation, business rule checks).
  - DAGs should be configurable and reusable across different pipelines.
- **Apache Spark**: Data processing engine for performing scalable data quality checks.
  - Use Spark DataFrames for data validation.
  - Distributed execution across multiple nodes for handling large datasets.
- **S3**: Store the raw data, logs, and quality check reports.
- **Ranger**: Define access control policies, ensuring only authorized users can modify or view data validation reports.

3.2. Data Quality Rule Configuration

Configuration Table (stored in a database or as JSON files in S3):

Rule_ID	Rule_Description	Rule_Type	Check_Level	Error_Handling	Alert_Level	Active
1	Check for null values in column A	Pre-Ingestion	Table	Log/Alert	High	TRUE
2	Foreign key relationship check	Post-Ingestion	Table	Abort Job	Medium	TRUE
3	Ensure date is within the range	Post-Ingestion	Field	Skip	Low	TRUE
4	Price should be a positive value	Post-Ingestion	Field	Abort Job	High	TRUE

- **Rule_Type**: Pre-ingestion or post-ingestion.
- **Check_Level**: Field or table-level validation.
- **Error_Handling**: What happens when validation fails (abort the job, log error, or alert users).
- **Alert_Level**: Determines the priority of alerts (high, medium, low).

3.3. Data Quality Check Pipeline (Airflow DAG)

3.3.1. Airflow DAG Example

python
Copy code

```
from airflow import DAG
from airflow.operators.python_operator import PythonOperator
from datetime import datetime

def validate_schema():
    # Implement schema validation logic
    pass

def validate_business_rules():
    # Implement business rule checks
    pass

def log_errors():
    # Logic to log errors and raise alerts
    pass

with DAG(dag_id='data_quality_validation',
start_date=datetime(2023, 1, 1), schedule_interval='@daily')
as dag:

    schema_validation = PythonOperator(
        task_id='schema_validation',
        python_callable=validate_schema
    )

    business_rule_check = PythonOperator(
        task_id='business_rule_check',
        python_callable=validate_business_rules
    )

    log_errors_task = PythonOperator(
        task_id='log_errors',
        python_callable=log_errors
    )

    schema_validation >> business_rule_check >>
log_errors_task
```

3.3.2. DAG Components

- **validate_schema()**: Function to check schema correctness.
- **validate_business_rules()**: Function to check business logic validation.
- **log_errors()**: Logs errors into a centralized location and raises alerts if necessary.

4. Detailed Quality Check Implementation

4.1. Schema Validation

- Ensure that the incoming data matches the defined schema.
- Schema fields are loaded from configuration files, and any deviation triggers an error.

4.2. Null Value Checks

- Identify any `NULL` values in non-nullable columns.
- Implement this check using Spark's `isNull()` function.

python
Copy code
```
def null_value_check(dataframe, columns):
    for column in columns:
        if
dataframe.filter(dataframe[column].isNull()).count() > 0:
            raise ValueError(f"Null values found in column
{column}")
```

4.3. Business Rule Validation

- Validate that business-specific rules are adhered to. Example: ensure that a product price is positive.

python
Copy code
```
def validate_price(dataframe):
    invalid_rows = dataframe.filter(dataframe['price'] < 0)
    if invalid_rows.count() > 0:
        raise ValueError("Negative prices found!")
```

4.4. Record Count Validation

- Ensure that the record counts between source and staging match.

python
Copy code
```
def record_count_validation(source_count, staging_count):
    if source_count != staging_count:
        raise ValueError(f"Record count mismatch! Source:
{source_count}, Staging: {staging_count}")
```

5. Monitoring and Logging

- **Centralized Logs**: Store all logs in AWS CloudWatch or ELK Stack.
- **Failure Notifications**: Implement Slack or email alerts for failed DQ checks using Airflow's notification system.

python
Copy code
```
from airflow.operators.email_operator import EmailOperator

alert_email = EmailOperator(
    task_id='send_failure_email',
    to='data_team@example.com',
    subject='Data Quality Check Failed',
```

```
    html_content="""<h3>Data Quality Check Failed</
h3><p>Please review the logs for more details.</p>"""
)
```

6. Reporting

6.1. Data Quality Reports

- Store reports on S3, accessible by authorized users via Ranger.
- Include details such as rule ID, validation status, error count, and action taken.

6.2. Alert Dashboard

- A real-time dashboard (e.g., using Grafana) to monitor data quality status and alerts.
- Integration with a logging system (e.g., ELK Stack) to visualize the logs and validation errors.

7. Security and Access Control

7.1. Ranger Policies

- Define access policies for who can view or modify data quality reports.
- Role-based access controls (RBAC) for data engineers, data analysts, and other stakeholders.

7.2. Encryption

- Data quality logs should be encrypted at rest in S3 using SSE.
- Encrypt data during transit using TLS.

8. Resource Requirements

- **Apache Spark**: Ensure enough cluster resources are available to process large datasets.
- **Airflow Scheduler**: Ensure Airflow DAGs can run at regular intervals without bottlenecks.
- **S3 Storage**: Adequate storage should be provisioned for logs and validation reports.

This detailed document should provide a clear roadmap for developers to start building the **Data Quality and Validation Service** for your modern data platform. The next steps would involve assigning roles, scheduling development tasks, and writing the corresponding tests for each feature.

1. Overview of Data Quality and Validation Service

The Data Quality (DQ) and Validation Service ensures the integrity, accuracy, and completeness of data before it is processed or moved to the next stage in the data pipeline. This service will be automated using **Autosys** for job scheduling and **Spark** for scalable performance, with **HDFS** for storage and **Hive** for reporting.

Key Objectives:

- **Data Integrity**: Ensure data complies with business rules and meets quality standards.

- **Automated Validation**: Execute quality checks automatically at different stages of the data pipeline.
- **Error Reporting**: Provide detailed reports on data quality failures with error logs for remediation.
- **Scalability**: Handle large data volumes and support batch, micro-batch, and streaming data validation.
- **Real-time Monitoring**: Ensure timely identification of data quality issues.

2. Functional Requirements

2.1. Pre-Ingestion Quality Checks

- Schema validation (data type, length, and mandatory fields).
- Duplication checks (check for duplicate records).
- Reference data validation (checking for valid lookup values).
- Null value checks for non-nullable fields.

2.2. Post-Ingestion Quality Checks

- Data format validation (ensure correct file format - e.g., Parquet/CSV).
- Consistency checks (e.g., foreign key relationships, range values).
- Business rule validation (e.g., price must be positive, date should be in a valid range).
- Record count validation between ingestion source and data platform.

2.3. Data Profiling

- Generate statistics such as max, min, average, and record count.
- Provide insight into distributions, cardinality, and anomalies.

2.4. Error Logging & Notification

- Log errors to a centralized system (e.g., HDFS or ELK Stack).
- Generate detailed error reports with error type, failing record details, and recommended remediation.
- Send alerts to defined stakeholders when data quality rules fail (using Autosys alerts, email, etc.).

2.5. Retrying Failed Data Loads

- Automatic retry mechanism in case of transient issues.
- Manual retry capability after data issues are fixed.

3. Technical Design

3.1. Architecture

- **Autosys**: Job scheduling for the quality checks pipeline.
    - Each Autosys job will consist of different steps (e.g., schema validation, business rule checks).
    - Jobs should be configurable and reusable across different pipelines.
- **Apache Spark**: Data processing engine for performing scalable data quality checks.
    - Use Spark DataFrames for data validation.

    - Distributed execution across multiple nodes for handling large datasets.
- **HDFS**: Store raw data, logs, and quality check reports.
- **Hive**: Store the results of data quality checks for querying and reporting purposes.
- **Ranger**: Define access control policies, ensuring only authorized users can modify or view data validation reports.

3.2. Data Quality Rule Configuration

Configuration Table (stored in Hive or as JSON files in HDFS):

Rule_ID	Rule_Description	Rule_Type	Check_Level	Error_Handling	Alert_Level	Active
1	Check for null values in column A	Pre-Ingestion	Table	Log/Alert	High	TRUE
2	Foreign key relationship check	Post-Ingestion	Table	Abort Job	Medium	TRUE
3	Ensure date is within the range	Post-Ingestion	Field	Skip	Low	TRUE
4	Price should be a positive value	Post-Ingestion	Field	Abort Job	High	TRUE

- **Rule_Type**: Pre-ingestion or post-ingestion.
- **Check_Level**: Field or table-level validation.
- **Error_Handling**: What happens when validation fails (abort the job, log error, or alert users).
- **Alert_Level**: Determines the priority of alerts (high, medium, low).

3.3. Data Quality Check Pipeline (Autosys Job Flow)

3.3.1. Autosys Job Flow Example

1. **Step 1**: Pre-ingestion Schema Validation Job
    - Runs the Spark job to check schema validity.
2. **Step 2**: Business Rule Validation Job
    - Checks the data against business-specific rules (e.g., price > 0).
3. **Step 3**: Error Logging & Report Generation Job
    - Logs errors and generates a detailed quality report.
4. **Step 4**: Alerts & Notification Job
    - Sends out notifications to stakeholders via email/alerts if validation fails.

4. Detailed Quality Check Implementation

4.1. Schema Validation

- Ensure that the incoming data matches the defined schema.

- Schema fields are loaded from configuration files, and any deviation triggers an error.

4.2. Null Value Checks

- Identify any `NULL` values in non-nullable columns.
- Implement this check using Spark's `isNull()` function.

python
Copy code
```
def null_value_check(dataframe, columns):
    for column in columns:
        if
dataframe.filter(dataframe[column].isNull()).count() > 0:
            raise ValueError(f"Null values found in column
{column}")
```

4.3. Business Rule Validation

- Validate that business-specific rules are adhered to. Example: ensure that a product price is positive.

python
Copy code
```
def validate_price(dataframe):
    invalid_rows = dataframe.filter(dataframe['price'] < 0)
    if invalid_rows.count() > 0:
        raise ValueError("Negative prices found!")
```

4.4. Record Count Validation

- Ensure that the record counts between source and staging match.

python
Copy code
```
def record_count_validation(source_count, staging_count):
    if source_count != staging_count:
        raise ValueError(f"Record count mismatch! Source:
{source_count}, Staging: {staging_count}")
```

5. Monitoring and Logging

- **Centralized Logs**: Store all logs in HDFS or ELK Stack.
- **Failure Notifications**: Implement notification mechanism in Autosys to alert teams of failed data quality checks.

python
Copy code
```
# Example: Autosys Failure Email Notification

insert_job: validate_schema
description: "Schema Validation Failure Alert"
alarm_if_fail: Y
notification: team_email@example.com
```

6. Reporting

6.1. Data Quality Reports

- Store reports in Hive, accessible via Hive queries.
- Include details such as rule ID, validation status, error count, and action taken.

6.2. Alert Dashboard

- A real-time dashboard (e.g., using Grafana) to monitor data quality status and alerts.
- Integration with a logging system (e.g., ELK Stack) to visualize the logs and validation errors.

7. Security and Access Control

7.1. Ranger Policies

- Define access policies for who can view or modify data quality reports in Hive or HDFS.
- Role-based access controls (RBAC) for data engineers, data analysts, and other stakeholders.

7.2. Encryption

- Data quality logs should be encrypted at rest in HDFS.
- Encrypt data during transit using TLS.

8. Resource Requirements

- **Apache Spark**: Ensure enough cluster resources are available to process large datasets.
- **Autosys Scheduler**: Ensure Autosys jobs can run at regular intervals without bottlenecks.
- **HDFS Storage**: Adequate storage should be provisioned for logs and validation reports.

This updated design for the **Data Quality and Validation Service** should provide the necessary guidelines for your team to start building the service with **Autosys** as the scheduler, **HDFS** for storage, and **Hive** for data query/reporting. Let me know if you need more adjustments!

Data Quality and Validation Service - Detailed Design Document

1. Introduction

This document details the design for the **Data Quality and Validation Service**, focusing on traceability from functional requirements to design elements, non-functional requirements, and the system's architecture. The service will ensure that data is accurate, valid, and consistent across different stages of the data pipeline.

2. Scope

This service will:

- Validate data before ingestion and after ingestion into the data platform.

- Automate quality checks using **Autosys** for job scheduling and **Spark** for validation processing.
- Store data and logs in **HDFS** and **Hive**.
- Report and track data quality issues via logs and alerts.

3. Functional Requirements

FR No.	Description	Priority	Acceptance Criteria
FR-001	Pre-ingestion Schema Validation	High	Schema must match the expected format; invalid records will be logged and reported.
FR-002	Null Value Checks for Mandatory Fields	High	All mandatory fields must be non-null. Null values should trigger an alert and log entry.
FR-003	Data Type and Format Validation	Medium	Data must be of the correct type (e.g., integer, string). Invalid data types are logged.
FR-004	Business Rule Validation (Price > 0, Date within range)	High	Business rules must be adhered to, and errors should be logged and reported immediately.
FR-005	Post-Ingestion Record Count Validation	High	Record count in the source and staging tables should match. Differences should trigger alerts.
FR-006	Error Reporting and Logging	High	Errors must be logged in HDFS or a centralized logging system and sent as email alerts.
FR-007	Data Profiling and Statistics Generation	Medium	Profiling statistics (e.g., max, min, avg) should be generated and stored in Hive.
FR-008	Retry Mechanism for Failed Jobs	Medium	Jobs that fail due to transient issues must be retried automatically.

4. Non-Functional Requirements (NFR)

NFR No.	Description	Priority	Acceptance Criteria
NFR-001	**Scalability**: Must handle growing data volumes	High	System must scale to validate petabyte-scale data with sub-minute processing times.
NFR-002	**Performance**: Timely data validation	High	The system should validate data within 5% of the batch window to avoid bottlenecks.
NFR-003	**Availability**: Must operate with high availability	High	99.9% uptime; Autosys jobs should be able to handle failovers automatically.
NFR-004	**Security**: Data and logs must be secured	High	Ranger policies must be enforced; all data at rest (HDFS, Hive) and in transit must be encrypted.
NFR-005	**Error Handling**: Effective retry and alert mechanism	Medium	Job retries must succeed 90% of the time after fixing transient issues.
NFR-006	**Monitoring and Logging**: Centralized monitoring	High	The system must log all errors and alert stakeholders with less than 2 minutes delay.
NFR-007	**Maintainability**: Code modularity and configuration	Medium	The system should be easily extendable for future quality rules. All rules should be configurable.

5. Detailed Design and Architecture

5.1. Architectural Overview

System Components:

- **Autosys**: Handles the orchestration and scheduling of data quality checks.
- **Spark**: Performs the actual validation logic at scale, processing data on distributed nodes.
- **HDFS**: Stores raw data, validation logs, and error reports.
- **Hive**: Stores summarized reports and data profiling statistics for querying by analysts.
- **Ranger**: Implements role-based access control for secured data access.

Data Flow:

1. **Pre-Ingestion**: Data lands in HDFS. Autosys triggers a Spark job to perform schema validation.
2. **Post-Ingestion**: After data is staged, Autosys triggers additional Spark jobs to validate data quality rules such as business constraints, format checks, and record count validations.
3. **Error Handling**: Errors identified during validation are logged and alerts are sent to stakeholders via email/notification systems integrated with Autosys.
4. **Reporting**: All validation results, including profiling statistics, are stored in Hive for downstream querying and reporting.

5.2. Data Quality Rule Engine

- **Rule Configuration**:
    - Data quality rules will be stored in a Hive table or as configuration files (JSON/XML) in HDFS.
    - Example Rule Configuration:
      json
      Copy code

```json
{
  "rule_id": "R001",
  "rule_type": "Pre-Ingestion",
  "rule_description": "Check for null values in mandatory fields",
  "validation_type": "null_check",
  "columns": ["column_a", "column_b"],
  "error_handling": "log_and_alert",
  "alert_level": "high"
}
```

- **Rule Application**:
    - Rules are applied at various stages of data processing. Each rule type corresponds to a Spark-based validation job that processes data in parallel.
    - Validation results are categorized into success, warning, and failure. All failures generate an alert.

5.3. Spark Job Flow

1. **Schema Validation**:
   - Reads the schema definition from Hive or JSON files.
   - Validates that incoming data matches the schema (e.g., data types, field names).
2. **Null Checks**:
   - For each mandatory column, check for null values and log any invalid rows.
   - Use the `isNull()` function in Spark:
3. python
   Copy code

```
   def null_value_check(df, columns):
4.     for col in columns:
5.         if df.filter(df[col].isNull()).count() > 0:
6.             raise ValueError(f"Null values found in
   {col}")
7.
```

8. **Business Rule Validation**:
   - For business-specific rules, such as checking that prices are positive, the Spark job will apply rules dynamically based on the configuration.
9. python
   Copy code

```
    def validate_business_rules(df):
10.     invalid_rows = df.filter(df['price'] <= 0)
11.     if invalid_rows.count() > 0:
12.         raise ValueError("Invalid prices detected!")
13.
```

14. **Record Count Validation**:
    - Ensure that the number of records matches between the source and the data platform:
15. python
    Copy code

```
    def record_count_check(source_count, platform_count):
16.     if source_count != platform_count:
17.         raise ValueError("Record count mismatch!")
```

6. Traceability Matrix

Requirement ID	Description	Implementation	Verification
FR-001	Schema validation pre-ingestion	Spark job reads schema from Hive or JSON and validates data	Schema validation success in log file
FR-002	Null value checks on mandatory fields	Spark job filters for null values and logs errors	Error log entries for null values
FR-003	Data type and format	Spark checks data types against	Logs generated for
FR-004	Business rule	Dynamic validation using rules from	Business rule
FR-005	Record count validation	Spark job compares record counts between source and platform	Record count matching log
FR-006	Error logging and	All errors logged to HDFS or a	Centralized logs and
FR-007	Data profiling	Spark generates summary statistics for	Data profiling report
FR-008	Retry mechanism	Autosys re-runs failed jobs if transient	Job retry success in

7. Non-Functional Traceability Matrix

NF	Description	Implementation	Verification
NFR-001	System must scale to handle large data sets	Autosys orchestrates Spark jobs across distributed nodes	Spark job execution time remains within threshold
NFR-002	Data validation must happen quickly	Spark processes jobs in parallel across cluster	Data validation speed (logs)
NFR-003	System must be highly available	Autosys configured for high availability and failover	Autosys uptime logs
NFR-004	Data and logs must be secure	Ranger policies enforce access control	Security audit logs from Ranger
NFR-005	System must handle transient failures gracefully	Autosys re-runs failed jobs and triggers alerting	Logs indicate success after retry

1. Introduction

This document provides a detailed design for a **Modern Data Platform** that allows efficient data ingestion, processing, and querying using federated data technologies. The platform is metadata-driven to ensure reusability across different applications. It integrates scheduling, processing, querying, and security services.

2. Scope

The platform provides:

- **Data Ingestion**: Scalable ingestion of data from multiple sources using Spark and Airflow.
- **Data Processing**: Transformation and validation of data using Spark with data stored in Iceberg-backed tables on S3.
- **Federated Querying**: Use of Starburst to query data across various systems.
- **Security and Access Control**: Role-based access via Apache Ranger.
- **Scheduling and Orchestration**: Airflow schedules the workflows and manages dependencies.
- **Reusability**: A metadata-driven approach for dynamic table generation, validations, and querying.

3. Functional Requirements

FR No.	Description	Priority	Acceptance Criteria
FR-00	Ingest data into S3 using Spark	High	Data should be ingested in predefined formats and partitioned using metadata configurations.
FR-00	Manage Iceberg table formats, partitioning, and compression	High	Data should be written into Iceberg tables using the formats and partitions defined in config.
FR-00	Schedule and orchestrate jobs using Airflow	High	Airflow DAGs should manage the scheduling and dependencies of Spark ingestion/processing jobs.
FR-00	Federated queries on Starburst across multiple data sources	High	Starburst should query federated sources (Iceberg, S3, Hive) with performance optimizations.
FR-00	Apply role-based access control using Apache Ranger	High	Data access should be governed using Ranger, based on defined roles and policies.
FR-00	Use metadata to drive table creation and data transformation	Medium	Metadata configurations should define the table schemas, partitioning rules, and business validations.

4. Non-Functional Requirements (NFR)

NFR	Description	Priority	Acceptance Criteria
NFR-0	**Scalability**: Must handle large-scale data sets	High	Platform should process petabyte-scale data with linear scalability using Spark and Iceberg.
NFR-0	**Performance**: Low-latency querying with Starburst	High	Starburst queries must return results in under 3 seconds for typical queries.
NFR-0	**Security**: Role-based access enforcement using Ranger	High	Ranger should enforce access policies with no unauthorized access to any dataset.
NFR-0	**Reliability**: Scheduled jobs should execute reliably	High	Airflow DAG success rate must be over 99.5% with automatic retries for transient issues.
NFR-0	**Maintainability**: Easily configurable and extendable	Medium	New data sources should be easily onboarded with minimal code changes by updating metadata.
NFR-0	**Availability**: The platform should be available 24/7	High	Downtime should be less than 1 hour per month, including planned maintenance.

5. Detailed Design and Architecture

5.1. High-Level Architecture

- **Data Ingestion**: Data from various sources is ingested into S3 using Spark jobs. Airflow orchestrates these jobs.
- **Data Storage**: Data is stored in **Iceberg** tables on S3, supporting incremental data updates, time travel, and partitioning.
- **Data Processing**: Spark jobs process, transform, and validate data, making use of configurations defined in metadata tables.
- **Federated Querying**: Starburst queries the data across Iceberg tables in S3 and federates data from other systems.
- **Security**: Apache Ranger enforces role-based access control for data at rest in S3 and during queries executed via Starburst.

5.2. Components and Design Details

1. **Metadata-Driven Framework**:

    - **Metadata Tables (in Hive)**: Define schemas, validation rules, partitioning, and compression formats.

    - **Example Metadata Table (schema_config)**:
      sql
      Copy code

```sql
CREATE TABLE metadata.schema_config (
    table_name STRING,
    schema_definition STRING,
    partition_columns ARRAY<STRING>,
    compression_type STRING,
    validation_rules STRING,
    data_format STRING
);
```

    - **JSON Configuration Example**:
      json
      Copy code

```json
{
  "table_name": "customer_data",
  "schema": {
    "customer_id": "string",
    "name": "string",
    "age": "int",
    "signup_date": "date"
```

```
  },
  "partition_columns": ["signup_date"],
  "compression": "snappy",
  "validations": {
    "age": {
      "type": "range",
      "min": 18,
      "max": 100
    }
  }
}

```

2. **Data Ingestion and Processing (Spark Jobs)**:
   - **Ingestion Flow**:
     - **Input**: Data from source systems (e.g., CSV, JSON, database).
     - **Output**: Iceberg tables on S3, with partitioning and compression as defined by metadata.
   - **Sample Spark Job Code**:

python
Copy code

```python
from pyspark.sql import SparkSession
from pyspark.sql.functions import col
import json

def load_metadata(table_name):
    # Read metadata configuration
    metadata_path = f"s3://metadata/{table_name}.json"
    with open(metadata_path, 'r') as f:
        return json.load(f)

def validate_data(df, validation_rules):
    # Example validation for age range
    age_rules = validation_rules.get("age", {})
    if age_rules:
        df = df.filter((col("age") >= age_rules["min"]) & (col("age") <= age_rules["max"]))
    return df

```

```python
def ingest_data(spark, source_path, table_name):
    metadata = load_metadata(table_name)
    schema = metadata["schema"]
    df = spark.read.format(metadata["data_format"]).load(source_path)

    # Apply validations
    df = validate_data(df, metadata["validations"])

    # Write to Iceberg table in S3
    df.write.format("iceberg").option("compression", metadata["compression"]).partitionBy(metadata["partition_columns"]).save(f"s3://data-lake/{table_name}")

spark = SparkSession.builder.appName("DataIngestion").getOrCreate()
ingest_data(spark, "s3://input-data/customers.csv", "customer_data")
```

3. **Orchestration (Airflow)**:

   - **DAG Design**: Airflow DAGs schedule and manage dependencies of Spark jobs. DAGs are dynamically generated based on metadata.
   - **Sample Airflow DAG Code**:
     python
     Copy code

```python
from airflow import DAG
from airflow.providers.apache.spark.operators.spark_submit import SparkSubmitOperator
from datetime import datetime

with DAG(dag_id="data_ingestion_pipeline",
         start_date=datetime(2023, 9, 1),
         schedule_interval="0 12 * * *") as dag:

    ingest_customer_data = SparkSubmitOperator(
        task_id="ingest_customer_data",
```

```
            application="s3://scripts/ingest_data.py",
            application_args=["customer_data"],
            conn_id="spark_default",
            name="CustomerDataIngestion"
        )

        ingest_customer_data

```

4. **Federated Queries (Starburst)**:
   - Starburst allows users to query across different systems, including Iceberg and S3. Queries are federated and optimized for performance.
   - **Example Query**:
     sql
     Copy code

```sql
SELECT customer_id, name, age
FROM iceberg.catalog.customer_data
WHERE signup_date > '2023-01-01';

```

5. **Security (Apache Ranger)**:
   - **Ranger Policies**: Define role-based access control for the data. Users are assigned roles that determine their permissions on various tables.
   - **Example Ranger Policy**:
     json
     Copy code

```json
{
  "policyName": "CustomerDataPolicy",
  "resource": {
    "database": "iceberg",
    "table": "customer_data"
  },
  "allowConditions": [
    {
      "users": ["analyst_user"],
      "permissions": ["SELECT"]
```

- `    },`
- `    {`
- `      "users": ["admin_user"],`
- `      "permissions": ["ALL"]`
- `    }`
- `  ]`
- `}`
-

6. Traceability Matrix (continued)

Requirement	Description	Implementation	Verification
FR-001	Ingest data into S3 using Spark	Spark job with metadata-driven ingestion	Unit tests for schema validation and data ingestion with
FR-002	Manage Iceberg table formats and	Spark jobs with Iceberg table creation, compression, and	Integration testing with Iceberg partitions and compression
FR-003	Schedule and orchestrate jobs using	Airflow DAGs scheduling Spark ingestion and processing jobs	Unit and integration tests to ensure DAGs run as scheduled
FR-004	Federated queries on Starburst across	Starburst with configured catalogs for Iceberg, S3, and	Performance testing on federated queries and ensuring query
FR-005	Apply role-based access control using	Ranger policies configured to enforce user roles and data	Validation of access control via security testing tools, and
FR-006	Metadata-driven table creation and	Metadata tables in Hive (or JSON) used for schema,	Unit tests for metadata-driven workflows and checking schema
NFR-001	Scalability for large-scale data processing	Distributed processing using Spark and partitioned Iceberg	Stress testing with large datasets and monitoring for resource
NFR-002	Low-latency querying with	Query optimization via Starburst and federated catalogs	Performance tests ensuring sub-3-second query responses
NFR-003	Role-based access using Ranger	Ranger policies integrated with Ranger UI and metadata	Security tests verifying proper access restrictions per user role
NFR-004	High availability and reliability	Redundant Airflow DAGs and retry mechanisms for Spark jobs	Availability tests with simulated failures, monitoring retry
NFR-005	Maintainability and ease of extension	Metadata-driven framework for onboarding new data sources	Review new data source onboarding steps, and regression
NFR-006	24/7 availability with minimal downtime	Deployment of highly available services for Spark, Airflow, and	Monitoring and alerting systems in place, with verification via

7. Sample Code and Configurations

7.1. Sample Metadata-Driven Config Table

This is stored in a Hive/SQL database and queried to define data ingestion, transformation, and storage logic.

sql
Copy code
```
CREATE TABLE metadata.config_table (
    source_system STRING,
    source_path STRING,
    target_table STRING,
    target_path STRING,
    partition_columns ARRAY<STRING>,
    compression_type STRING,
    validation_rules STRING
);
```
7.2. Spark Ingestion Script (Python)

This code demonstrates metadata-driven ingestion with validations and data partitioning for Iceberg tables.

python
Copy code
```
from pyspark.sql import SparkSession
from pyspark.sql.functions import col
import json

def load_metadata(table_name):
    # Load metadata config from S3 or Hive
    metadata_path = f"s3://metadata/{table_name}.json"
    with open(metadata_path, 'r') as f:
        return json.load(f)

def validate_data(df, validation_rules):
    # Example: age range validation
    if 'age' in validation_rules:
        min_age, max_age = validation_rules['age']['min'],
validation_rules['age']['max']
        df = df.filter((col("age") >= min_age) & (col("age")
<= max_age))
    return df

def ingest_data(spark, table_name, source_path):
    metadata = load_metadata(table_name)

    # Load data based on schema
    df = spark.read.format("csv").option("header",
"true").load(source_path)

    # Apply validation rules
    df = validate_data(df, metadata["validations"])
```

```
    # Write data to Iceberg tables with defined partitioning
and compression

df.write.format("iceberg").partitionBy(metadata["partition_co
lumns"]).option("compression",
metadata["compression"]).save(f"s3://data-lake/
{metadata['target_table']}")

spark =
SparkSession.builder.appName("IngestionPipeline").getOrCreate
()
ingest_data(spark, "customer_data", "s3://input-data/
customers.csv")
```

7.3. Airflow DAG Example (Python)

The DAG schedules the Spark ingestion process, which uses metadata-driven configurations.

python
Copy code

```
from airflow import DAG
from airflow.providers.apache.spark.operators.spark_submit
import SparkSubmitOperator
from datetime import datetime

default_args = {
    'owner': 'data_team',
    'start_date': datetime(2024, 1, 1),
    'retries': 3,
}

with DAG(dag_id="customer_data_ingestion",
         default_args=default_args,
         schedule_interval="@daily") as dag:

    spark_submit = SparkSubmitOperator(
        task_id="run_customer_data_ingestion",
        application="s3://scripts/ingest_data.py",
        application_args=["customer_data", "s3://input-data/
customers.csv"],
        conn_id="spark_default",
        name="customer_data_ingestion"
    )

    spark_submit
```

7.4. Starburst Query Example (SQL)

This query demonstrates federated data access from Iceberg tables and other sources.

sql
Copy code
```sql
SELECT customer_id, name, signup_date
FROM iceberg.catalog.customers
WHERE signup_date > '2024-01-01';
```

7.5. Apache Ranger Policy

This JSON policy defines access control for querying customer data in Iceberg.

json
Copy code
```json
{
  "policyName": "CustomerDataAccessPolicy",
  "resource": {
    "database": "iceberg",
    "table": "customer_data"
  },
  "allowConditions": [
    {
      "users": ["data_analyst"],
      "permissions": ["SELECT"]
    },
    {
      "users": ["data_admin"],
      "permissions": ["ALL"]
    }
  ]
}
```

8. Conclusion

This design document outlines the architecture of the modern data platform with metadata-driven ingestion, transformation, and querying capabilities. The traceability matrix ensures that all functional and non-functional requirements are met. The sample code and configuration sections provide developers with a starting point for implementation, enabling scalable, secure, and efficient data workflows.

Data Quality Services Design Document for Modern Data Platform

1. Introduction

The Data Quality Service is a critical component of the modern data platform, ensuring that data integrity, consistency, and reliability are maintained across the three layers: Staging, Standardization, and Consumption. This document provides a detailed design for implementing Data Quality Services, with a focus on functional and non-functional requirements, a traceability matrix, and metadata-driven automation to support a reusable framework for onboarding and managing data across the platform.

2. Platform Architecture Overview

The modern data platform consists of:

- **S3 (or HDFS)** as the data lake for storing raw and processed data.
- **Iceberg** for table management, partitioning, and versioning.
- **Airflow (or AutoSys)** for job orchestration and scheduling.
- **Spark** for distributed data processing and transformations.
- **Starburst** for federated data queries across various data sources.
- **Apache Ranger** for access control and security enforcement.
- **Three-layer structure**:
  - **Staging Layer**: Raw data from source systems is ingested.
  - **Standardization Layer**: Data is cleaned, validated, and transformed into a standard schema.
  - **Consumption Layer**: Processed data is made available for querying and reporting.

3. Functional Requirements

ID	Description
F R-	Data must be ingested into the platform via metadata-driven Spark jobs, ensuring data schema and validation rules are defined in configuration files (JSON or metadata tables).
F R-	Validation rules such as null checks, uniqueness, data type consistency, and custom business rules must be enforced in the Staging and Standardization layers.
F R-	Data Quality metrics (e.g., number of null values, invalid entries, and duplicates) should be automatically calculated and stored in a monitoring table for each dataset.
F R-	Alerts and notifications must be triggered if any data quality thresholds (configurable) are breached.
F R-	Data Quality validations must run as part of Airflow DAGs (or AutoSys jobs), ensuring that no data moves to the Standardization or Consumption layer without passing the defined quality
F R-	The platform should support multiple data formats (CSV, JSON, Avro, Parquet) with metadata-driven transformations.
F R-	Failed data records must be logged and stored in an exception table for later review, with detailed logs available.
F R-	Provide a report of data quality metrics for every run to Data Stewards and Administrators for manual review and remediation.
F R-	Data lineage tracking must be implemented to trace each dataset's origin from the Staging to the Consumption layer.
F R-	Role-based access to data quality reports and results must be enforced through Apache Ranger, allowing only authorized personnel to view and edit data quality metrics.

4. Non-Functional Requirements

ID	Description
NF R-0	The Data Quality service must scale to handle large volumes of data across multiple sources with minimal performance impact on ingestion and transformation processes.

NFR-0	All data quality checks must execute within defined SLAs, typically ensuring validation and reporting within 5-10 minutes for each dataset.
NFR-0	The service must ensure data availability 99.99% of the time, ensuring that the platform is fault-tolerant and can handle transient errors.
NFR-0	The platform should support horizontal scaling by distributing data quality checks across Spark clusters to reduce processing time.
NFR-0	Data Quality validation and monitoring jobs should have retry and failover mechanisms for high availability.
NFR-0	The Data Quality Service must maintain detailed audit logs of every validation run, capturing data volumes, failures, and any issues encountered.
NFR-0	The system should be designed with security in mind, ensuring that data quality checks do not expose sensitive or restricted data during processing.
NFR-0	Data Quality metadata (e.g., validation configurations, monitoring results) must be stored in a highly available and durable store, such as a metadata database or a Hive metastore.
NFR-0	The system must support multi-tenant environments, ensuring data separation and role-based access for different applications and users.
NFR-0	The Data Quality service should have self-healing mechanisms, automatically restarting failed jobs where possible.

5. Metadata-Driven Framework

To ensure flexibility and reusability, the Data Quality service will be metadata-driven. Metadata will define the schema, validation rules, partitioning strategies, and transformation logic. The following components will drive the metadata framework:

- **Config Tables/JSON Files**:
  - A metadata table (e.g., stored in Hive or a relational DB) will contain all necessary information, including source/target paths, validation rules, and partitioning columns.
  - Alternatively, JSON config files can be used for defining validation rules, ensuring portability and ease of configuration for different applications.

6. Data Quality Layers

6.1. Staging Layer:

- In the Staging layer, raw data is ingested into the platform. Basic data quality checks (such as schema validation, null checks, and type validation) are applied. Failed records are stored in an exception table, with the details logged for debugging and remediation.

6.2. Standardization Layer:

- In the Standardization layer, the data undergoes more rigorous validation, including complex business rule validation, duplicate checks, and consistency checks across datasets. Once validated, the data is transformed into a standardized format and stored in Iceberg tables with appropriate partitioning and compression applied.

6.3. Consumption Layer:

- In the Consumption layer, the data is made available for querying and analysis. Data Quality metrics are generated at this stage and shared with stakeholders. Queries are run using Starburst to federate data from multiple sources, ensuring the integrity and consistency of the output data.

7. Traceability Matrix

Require ment	Description	Implementation	Verification
FR-001	Metadata-driven data ingestion	Config tables/JSON files for schema and validation logic	Unit tests validating schema configurations and ingestion
FR-002	Validation rules enforcement	Implement Spark jobs to apply validation rules on raw data	Automated tests for validation and error logging
FR-003	Data Quality metrics	Store metrics (nulls, duplicates)	Integration tests for correctness
FR-004	Trigger alerts for failed data checks	Alerts via Airflow/AutoSys and external services like Slack	Test alerts by simulating invalid data conditions
FR-005	Validation in Airflow DAGs/AutoSys jobs	Integrate validation checks into the orchestration framework	Review of DAG/job logs for successful validation
NFR-00 1	Scalability for large data volumes	Distributed processing using Spark and partitioned Iceberg	Stress testing with large datasets and resource monitoring
NFR-00 2	Low-latency validation	Parallel processing of data quality checks using Spark	Performance testing ensuring validation completes within
NFR-00	High availability and	Retry mechanisms and job	Simulate job failures and verify
NFR-00 4	Security of data quality checks	Implement Apache Ranger policies for data access	Security tests ensuring access control rules are enforced

8. Sample Code for Data Quality Checks

8.1. Spark Data Validation Script (Python)

python
Copy code

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import col, isnull, count

def validate_schema(df, expected_schema):
    # Validate data schema
    actual_schema = set(df.columns)
    missing_columns =
expected_schema.difference(actual_schema)
    if missing_columns:
        raise ValueError(f"Missing columns:
{missing_columns}")
    return df

def check_null_values(df, columns):
    # Check for null values
    null_counts = df.select([count(when(isnull(c),
c)).alias(c) for c in columns])
    return null_counts
```

```
def run_data_quality_checks(spark, config):
    # Load data
    df =
spark.read.format(config['format']).load(config['source_path'
])

    # Validate schema
    df = validate_schema(df, set(config['expected_schema']))

    # Run null checks
    null_report = check_null_values(df,
config['required_columns'])
    null_report.show()

    # Write data if validation passes

df.write.format("iceberg").mode("overwrite").save(config['tar
get_path'])

spark =
SparkSession.builder.appName("DataQualityChecks").getOrCreate
()
config = {
    "format": "csv",
    "source_path": "s3://data-lake/raw/customers.csv",
    "expected_schema": {"customer_id", "name", "age",
"email"},
    "required_columns": ["customer_id", "email"],
    "target_path": "s3://data-lake/processed/customers"
}
run_data_quality_checks(spark, config)
```

9. Conclusion

This detailed design document outlines the architecture and requirements

Data Quality Services Design Document for Modern Data Platform

1. Introduction

This document outlines the design for **Data Quality Services** within a modern data platform. The platform leverages S3 (or HDFS), Iceberg for versioned data, Airflow (or AutoSys) for orchestration, Spark for distributed data processing, Starburst for federated queries, and Apache Ranger for security and access control. Data quality is essential for ensuring that the data ingested, transformed, and consumed within the platform adheres to expected standards for accuracy, completeness, and integrity.

This document covers the functional and non-functional requirements, traceability matrix, detailed design, metadata-driven configuration, and sample code to guide the implementation of data quality services.

2. System Architecture

The data platform consists of three key layers:

1. **Staging Layer**: Raw data is ingested from external sources and placed into S3 (or HDFS). Basic data validation checks are performed here.
2. **Standardization Layer**: Data is transformed into a standardized format with strict validation rules enforced, ensuring data adheres to the required schema.
3. **Consumption Layer**: Processed data is made available for querying and analysis via Starburst or similar federated query engines.

Each layer interacts with multiple services, orchestrated through Airflow (or AutoSys), and ensures security policies are enforced via Apache Ranger.

3. Functional Requirements

ID	Description
FR-001	Data must be ingested into the platform via Spark jobs, with metadata defining validation rules, schema, and transformations.
FR-002	Validate source data based on schema conformity, null checks, data type validation, and custom business rules.
FR-003	Data Quality metrics must be calculated and logged, including record counts, null values, invalid data entries, and duplicates.
FR-004	Implement validation rules during each stage of the pipeline (Staging, Standardization, Consumption) to ensure data integrity.
FR-005	Data Quality validation results must be stored in a monitoring table, with alerts triggered if any validations fail.
FR-006	Exception handling must be built in to capture and log any records that fail validation and provide options for remediation.
FR-007	Role-based access controls for Data Quality reports, validation logs, and alerts must be enforced via Apache Ranger.
FR-008	Data Quality Service should support various data formats (CSV, JSON, Avro, Parquet).
FR-009	Data Quality checks should include lineage tracking to understand the flow of data from ingestion to consumption.
FR-010	Provide support for continuous Data Quality monitoring, ensuring periodic revalidation of data.

4. Non-Functional Requirements

ID	Description
NFR-001	The platform should be scalable, handling large volumes of data while minimizing performance degradation.

NFR -002	Validation checks must adhere to SLAs, ensuring completion within a maximum of 10 minutes for medium-sized datasets (up to 500 GB).
NFR -003	The platform should support fault-tolerance and retry mechanisms for job failures.
NFR -004	Data quality services should handle schema evolution without requiring major rewrites of validation code.
NFR -005	The solution should ensure data security by encrypting data both in transit and at rest.
NFR -006	The solution must comply with industry-standard regulations like GDPR or HIPAA, ensuring the protection of sensitive data.
NFR -007	The platform should maintain detailed audit logs for all validation checks, capturing metadata such as timestamps, validation type, and outcome.
NFR -008	Data Quality Service must operate 24x7 with minimal downtime, aiming for 99.9% availability.
NFR -009	System performance must remain stable even when concurrent validations are running across different datasets and applications.

5. Data Quality Service Design

5.1 Staging Layer

Purpose: Ingest raw data from source systems into the data platform. Perform basic data quality checks to ensure the integrity of incoming data.

- **Validation Rules**:
    - Schema conformity: Ensure incoming data matches the expected schema (columns, types).
    - Null value checks: Ensure that non-nullable fields are populated.
    - Basic data type validation: Ensure each field has the correct data type.
- **Actions**:
    - **Invalid Data Handling**: Store failed records in an exception table for further analysis and remediation.
    - **Logging**: Log validation failures, count the number of invalid records, and calculate the percentage of invalid entries for reporting.
    - **Schema Validation**: Dynamically check schema against the predefined metadata-driven configuration file (JSON or metadata table).

5.2 Standardization Layer

Purpose: Apply transformations to standardize the ingested data and enforce stricter data quality validation rules.

- **Validation Rules**:
    - Duplicate checks: Ensure that duplicate records are identified and handled according to business rules.
    - Cross-field validation: Ensure fields within the same row have consistent values.
    - Business rule validation: Implement validation for custom business rules specific to the dataset (e.g., date ranges, allowed values).

- **Actions**:
  - **Exception Handling**: Store records that fail validation in a quarantine zone for manual review or automated remediation.
  - **Auditing**: Capture validation metrics and store them in a Data Quality monitoring table.
  - **Lineage Tracking**: Maintain data lineage information to trace data back to its source system, ensuring full transparency.

5.3 Consumption Layer

Purpose: Ensure that the final, validated, and transformed data is available for query and reporting.

- **Validation Rules**:
  - Verify that the data conforms to the final consumption schema.
  - Run periodic data quality checks on the data in the Consumption layer to ensure continued data integrity.
- **Actions**:
  - **Data Availability**: Make validated data available in Iceberg tables, partitioned and compressed for optimized querying.
  - **Audit Reports**: Generate Data Quality reports that summarize the number of records ingested, transformed, and rejected at each stage.

6. Metadata-Driven Approach

The design is based on a metadata-driven framework, where configurations such as validation rules, schema, partitioning, and transformation logic are stored in external metadata tables or JSON files. This allows the system to dynamically adapt to different data sources and datasets without requiring code changes.

6.1 Configuration Schema

- **Schema Definition**:
  - Metadata tables will define the expected schema for each dataset (e.g., column names, data types, nullable fields).
- **Validation Rules**:
  - Each dataset will have its own validation rules defined in the metadata, including null checks, regex validation, and cross-field checks.

6.2 Sample Configuration (JSON)

json
Copy code

```
{
  "dataset_name": "customer_data",
  "staging_validation": {
    "schema": {
      "customer_id": "string",
      "name": "string",
      "age": "integer",
```

```
      "email": "string",
      "created_at": "timestamp"
    },
    "null_checks": ["customer_id", "email"],
    "regex_validation": {
      "email": "^\\w+@[a-zA-Z_]+?\\.[a-zA-Z]{2,3}$"
    }
  },
  "standardization_validation": {
    "duplicate_check": ["customer_id"],
    "business_rules": {
      "age_range": {
        "min": 18,
        "max": 99
      }
    }
  }
}
```

7. Traceability Matrix

Requirement ID	Description	Implementation	Verification
FR-001	Data ingestion via Spark with metadata configuration	Metadata-driven ingestion with config tables/JSON	Unit tests for schema conformity
FR-002	Schema and data validation	Schema validation and null checks in Spark jobs	Integration tests for validation rules
FR-003	Data Quality metric calculation	Spark jobs calculate null count,	Data Quality reports
FR-004	Validation in each layer	Validation applied at each	Review logs and
FR-007	Role-based access control	Apache Ranger enforcing	Security testing for
NFR-001	Scalability for large datasets	Spark for distributed	Performance

8. Sample Code for Data Quality

python
Copy code

```python
from pyspark.sql import SparkSession
from pyspark.sql.functions import col, count, when, lit

# Initialize Spark session
spark = SparkSession.builder \
    .appName("DataQualityService") \
    .getOrCreate()

# Configuration for data validation (from metadata)
```

```
config = {
    "source_path": "s3://data-lake/raw/customer_data",
    "target_path": "s3://data-lake/processed/customer_data",
    "schema": {
        "customer_id": "string",
        "name": "string",
        "age": "integer",
        "email": "string"
    },
    "null_check_columns": ["customer_id", "email"],
    "regex_validation": {
        "email": "^\\w+@[a-zA-Z_]+?\\.[a-zA-Z]{2,3}$"
    }
}

# Load the data
df = spark.read.csv(config["source_path"], header=True,
inferSchema=True)

# Null value validation
for column in config["null_check_columns"]:
    null_count = df.filter(col(column).isNull()).count()
    if null_count > 0:
        print(f"Validation Failed: {null_count} null values
found in {column}")

# Regex validation for email
invalid_emails =
df.filter(~col("email").rlike(config["regex_validation"]
["email"])).count()
if invalid_emails > 0:
    print(f"Validation Failed: {invalid_emails} invalid
emails found")

# Save validated data
df.write.mode("overwrite").parquet(config["target_path"])
```

9. Conclusion

This design document outlines a comprehensive approach for building a scalable and robust **Data Quality Service** for a modern data platform. By leveraging metadata-driven configurations, distributed processing with Spark, orchestration via Airflow, and federated queries via Starburst, this design ensures data quality at every stage of the pipeline while remaining flexible and scalable. The incorporation of detailed validation rules, lineage tracking, exception handling, and auditing makes this design suitable for enterprise-level applications.

Data Quality and Validation Service - README

1. Overview

The **Data Quality (DQ) and Validation Service** ensures that data is accurate, complete, and conforms to defined business rules before further processing in the data pipeline. This service is automated using **Apache Airflow** and **Apache Spark** for scalability, ensuring seamless integration with data pipelines.

Key Objectives:

- **Data Integrity**: Validate data against predefined rules and quality standards.
- **Automated Validation**: Automate quality checks at various stages of the data pipeline.
- **Error Reporting**: Generate detailed error reports to assist with remediation.
- **Scalability**: Handle large datasets efficiently, including batch, micro-batch, and streaming data.
- **Real-time Monitoring**: Provide early detection of data quality issues.

2. Functional Requirements

2.1. Pre-Ingestion Quality Checks

- **Schema Validation**: Ensure correct data types, field lengths, and mandatory fields.
- **Duplication Checks**: Identify and flag duplicate records.
- **Reference Data Validation**: Check for valid lookup or reference values.
- **Null Value Checks**: Ensure no NULL values exist in non-nullable fields.

2.2. Post-Ingestion Quality Checks

- **Data Format Validation**: Validate correct file formats (e.g., Parquet, CSV).
- **Consistency Checks**: Verify relationships (e.g., foreign keys, value ranges).
- **Business Rule Validation**: Ensure that specific business rules are respected (e.g., positive prices, valid date ranges).
- **Record Count Validation**: Ensure the number of records matches between the ingestion source and platform.

2.3. Data Profiling

- Generate data statistics (max, min, average, etc.).
- Identify anomalies, cardinality, and data distributions.

2.4. Error Logging & Notification

- Log all errors to a centralized system (e.g., AWS CloudWatch, ELK Stack).
- Generate error reports with detailed logs, including error types, failing records, and possible fixes.
- Send notifications to relevant stakeholders via Slack, email, etc.

2.5. Retrying Failed Data Loads

- Automatically retry on transient failures.
- Allow manual retries after errors are resolved.

3. Technical Design

3.1. Architecture

- **Apache Airflow**: Used to orchestrate quality checks using Directed Acyclic Graphs (DAGs) for reusable, configurable pipelines.
- **Apache Spark**: Provides distributed processing to ensure scalable data validation.
- **S3**: Used for storing raw data, validation logs, and reports.
- **Apache Ranger**: Handles access control for managing and viewing validation reports.

3.2. Data Quality Rule Configuration

Configuration tables (in a database or JSON files) specify the validation rules, including the rule type, error handling, and alert priority:

Rule_ID	Rule_Description	Rule_Type	Check_Level	Error_Handling	Alert_Level	Active
1	Check for null values in column A	Pre-Ingestion	Table	Log/Alert	High	TRUE
2	Foreign key relationship check	Post-Ingestion	Table	Abort Job	Medium	TRUE
3	Ensure date is within range	Post-Ingestion	Field	Skip	Low	TRUE
4	Price should be positive	Post-Ingestion	Field	Abort Job	High	TRUE

3.3. Data Quality Check Pipeline (Airflow DAG)

Example Airflow DAG:

python
Copy code

```
from airflow import DAG
from airflow.operators.python_operator import PythonOperator
from datetime import datetime

def validate_schema():
    # Implement schema validation logic
    pass

def validate_business_rules():
    # Implement business rule checks
    pass

def log_errors():
    # Logic to log errors and raise alerts
    pass
```

```
with DAG(dag_id='data_quality_validation',
start_date=datetime(2023, 1, 1), schedule_interval='@daily')
as dag:

    schema_validation = PythonOperator(
        task_id='schema_validation',
        python_callable=validate_schema
    )

    business_rule_check = PythonOperator(
        task_id='business_rule_check',
        python_callable=validate_business_rules
    )

    log_errors_task = PythonOperator(
        task_id='log_errors',
        python_callable=log_errors
    )

    schema_validation >> business_rule_check >>
log_errors_task
```

4. Detailed Quality Check Implementation

4.1. Schema Validation

Verify that incoming data matches the defined schema, using configuration files for comparison. Deviations trigger errors.

4.2. Null Value Checks

Check for NULL values in non-nullable fields using Spark's `isNull()` function.

python
Copy code
```
def null_value_check(dataframe, columns):
    for column in columns:
        if
dataframe.filter(dataframe[column].isNull()).count() > 0:
            raise ValueError(f"Null values found in column
{column}")
```

4.3. Business Rule Validation

Ensure business-specific rules are enforced, such as ensuring product prices are positive.

python
Copy code
```
def validate_price(dataframe):
    invalid_rows = dataframe.filter(dataframe['price'] < 0)
    if invalid_rows.count() > 0:
        raise ValueError("Negative prices found!")
```

4.4. Record Count Validation

Ensure that record counts between source and staging environments match.

python
Copy code
```
def record_count_validation(source_count, staging_count):
    if source_count != staging_count:
        raise ValueError(f"Record count mismatch! Source: 
{source_count}, Staging: {staging_count}")
```

5. Monitoring and Logging

- **Centralized Logs**: Store logs in AWS CloudWatch or the ELK Stack.
- **Failure Notifications**: Use Airflow's notification system to send alerts (e.g., email, Slack).

python
Copy code
```
from airflow.operators.email_operator import EmailOperator

alert_email = EmailOperator(
    task_id='send_failure_email',
    to='data_team@example.com',
    subject='Data Quality Check Failed',
    html_content="""<h3>Data Quality Check Failed</
h3><p>Please review the logs for more details.</p>"""
)
```

6. Reporting

6.1. Data Quality Reports

- Store data quality reports in S3, accessible through role-based policies defined in Ranger.
- Reports include rule details, validation status, and error counts.

6.2. Alert Dashboard

- Create real-time dashboards (e.g., with Grafana) to visualize data quality status and alerts.
- Integrate with centralized logging (e.g., ELK Stack) to track validation errors.

7. Security and Access Control

7.1. Ranger Policies

- Define role-based access controls (RBAC) for data engineers, analysts, and other stakeholders.

7.2. Encryption

- Data quality logs should be encrypted in S3 using server-side encryption (SSE).
- Use TLS to encrypt data during transit.

8. Resource Requirements

- **Apache Spark**: Ensure adequate cluster resources for large datasets.
- **Airflow Scheduler**: Ensure efficient scheduling for Airflow DAGs.
- **S3 Storage**: Provision enough storage for logs and validation reports.

This README serves as a reference for developers to understand the structure, objectives, and technical implementation of the Data Quality and Validation Service. It outlines key tasks and configuration to ensure successful deployment and operation of the service within the modern data platform.

Data Quality and Validation Service - README

1. Overview

The **Data Quality (DQ) and Validation Service** ensures that data is accurate, complete, and conforms to defined business rules before further processing in the data pipeline. This service is fully automated and designed to scale across various data workloads, ensuring smooth integration with the existing data infrastructure.

Key Objectives:

- **Data Integrity**: Validate data against predefined rules and quality standards.
- **Automated Validation**: Automate quality checks at various stages of the data pipeline.
- **Error Reporting**: Generate detailed error reports to assist with remediation.
- **Scalability**: Handle large datasets efficiently, including batch, micro-batch, and streaming data.
- **Real-time Monitoring**: Provide early detection of data quality issues.

2. Functional Requirements

2.1. Pre-Ingestion Quality Checks

- **Schema Validation**: Ensure correct data types, field lengths, and mandatory fields.
- **Duplication Checks**: Identify and flag duplicate records.
- **Reference Data Validation**: Check for valid lookup or reference values.
- **Null Value Checks**: Ensure no NULL values exist in non-nullable fields.

2.2. Post-Ingestion Quality Checks

- **Data Format Validation**: Validate correct file formats (e.g., Parquet, CSV).
- **Consistency Checks**: Verify relationships (e.g., foreign keys, value ranges).
- **Business Rule Validation**: Ensure that specific business rules are respected (e.g., positive prices, valid date ranges).
- **Record Count Validation**: Ensure the number of records matches between the ingestion source and platform.

2.3. Data Profiling

- Generate data statistics (max, min, average, etc.).
- Identify anomalies, cardinality, and data distributions.

2.4. Error Logging & Notification

- Log all errors to a centralized system for tracking and remediation.
- Generate error reports with detailed logs, including error types, failing records, and possible fixes.
- Send notifications to relevant stakeholders via defined communication channels (e.g., email, messaging platforms).

2.5. Retrying Failed Data Loads

- Automatically retry on transient failures.
- Allow manual retries after errors are resolved.

3. Technical Design

3.1. Architecture

- **Orchestration**: Data quality checks are managed and scheduled via a flexible, configurable workflow. Workflows consist of reusable tasks that can be applied across different stages of the data pipeline.
- **Processing Engine**: A distributed, parallel-processing system ensures that data quality checks are scalable and can handle large datasets efficiently.
- **Storage**: Centralized storage is used to store raw data, logs, and reports generated during data validation.
- **Access Control**: A role-based access control system ensures that only authorized users can view or modify validation reports.

3.2. Data Quality Rule Configuration

The data quality rules are stored in configuration tables (or JSON files) and specify what type of validation should be performed, the severity of any errors, and the handling of alerts:

Rule_ID	Rule_Description	Rule_Type	Check_Level	Error_Handling	Alert_Level	Active
1	Check for null values in column A	Pre-Ingestion	Table	Log/Alert	High	TRUE
2	Foreign key relationship check	Post-Ingestion	Table	Abort Job	Medium	TRUE
3	Ensure date is within range	Post-Ingestion	Field	Skip	Low	TRUE

4	Price should be positive	Post-Ingestion	Field	Abort Job	High	TRUE

3.3. Data Quality Check Workflow

Example Workflow Structure:

python
Copy code
```
# Pseudocode representation of the validation process

def validate_schema():
    # Implement schema validation logic
    pass

def validate_business_rules():
    # Implement business rule checks
    pass

def log_errors():
    # Logic to log errors and raise alerts
    pass

# Workflow: Schema Validation -> Business Rule Check -> Error
Logging
```

4. Detailed Quality Check Implementation

4.1. Schema Validation

Ensure that incoming data matches the defined schema, using configuration files for comparison. Any deviation triggers an error.

4.2. Null Value Checks

Check for NULL values in non-nullable fields using data processing functions that scan the data and identify non-compliant records.

python
Copy code
```
def null_value_check(dataframe, columns):
    for column in columns:
        if
dataframe.filter(dataframe[column].isNull()).count() > 0:
            raise ValueError(f"Null values found in column
{column}")
```

4.3. Business Rule Validation

Ensure business-specific rules are enforced, such as ensuring product prices are positive.

python
Copy code
```
def validate_price(dataframe):
    invalid_rows = dataframe.filter(dataframe['price'] < 0)
    if invalid_rows.count() > 0:
        raise ValueError("Negative prices found!")
```
4.4. Record Count Validation

Ensure that record counts between source and staging environments match.

python
Copy code
```
def record_count_validation(source_count, staging_count):
    if source_count != staging_count:
        raise ValueError(f"Record count mismatch! Source:
{source_count}, Staging: {staging_count}")
```

5. Monitoring and Logging

- **Centralized Logs**: Store logs in a central logging platform for easy tracking and auditing.
- **Failure Notifications**: Notifications will be sent via email or messaging channels if any quality checks fail.

python
Copy code
```
# Example code to send failure alerts
alert_email = EmailOperator(
    task_id='send_failure_email',
    to='data_team@example.com',
    subject='Data Quality Check Failed',
    html_content="""<h3>Data Quality Check Failed</
h3><p>Please review the logs for more details.</p>"""
)
```

6. Reporting

6.1. Data Quality Reports

- Reports are stored in a centralized location and include details such as rule ID, validation status, error counts, and actions taken.
- Only authorized users can access these reports through the system's access control settings.

6.2. Alert Dashboard

- A real-time dashboard can be created to track data quality status and issues.
- Integration with centralized logging and alerting systems allows users to visualize the errors and their statuses.

7. Security and Access Control

7.1. Access Control Policies

- Role-based access controls (RBAC) are implemented to ensure that only authorized personnel can access or modify data quality reports.
- Different roles can be assigned to stakeholders such as data engineers and analysts, ensuring appropriate access levels for each.

7.2. Encryption

- Data quality logs and reports are encrypted at rest.
- Data is encrypted during transmission to ensure the security and integrity of sensitive data.

8. Resource Requirements

- **Processing Engine**: Ensure adequate resources are provisioned to handle the data validation processes, especially for large datasets.
- **Workflow Scheduler**: Ensure that workflows are scheduled at regular intervals, avoiding bottlenecks.
- **Storage**: Adequate storage should be provisioned for logs, reports, and any large datasets involved in validation.

This README provides the necessary details for the development and implementation of the Data Quality and Validation Service within the modern data platform. It outlines the objectives, technical design, and implementation steps to ensure a scalable, reusable, and robust data quality solution.

1. Overview

The Iceberg table will store dimension data with **SCD Type 2** semantics. In SCD Type 2, we maintain both historical and current data by adding new rows for changes to dimensional attributes. Iceberg's features, such as partitioning, snapshot isolation, and schema evolution, make it an ideal choice for this purpose in a data lake architecture using S3 as storage.

2. Schema Design for SCD Type 2

2.1. Table Structure

The dimensional table must support the ability to track changes to records over time. The following columns are typically needed:

Column Name	Data Type	Description
`dimension_id`	BIGINT	Surrogate key for the dimension (primary key).
`natural_key`	STRING	Business key (used for identifying duplicates).
`attribute_1`	STRING	First attribute (can be any dimension attribute).

`attribute_2`	STRING	Second attribute (can be any dimension attribute).
`effective_date`	DATE	Start date of the record validity.
`expiry_date`	DATE	End date of the record validity (NULL if active).
`is_current`	BOOLEAN	Indicates if the record is the current version.
`created_at`	TIMESTAMP	Timestamp when the record was created.
`updated_at`	TIMESTAMP	Timestamp when the record was last updated.

Note:

- Iceberg tables support **schema evolution** without expensive rewrites, meaning we can add or modify columns in the future without breaking the existing schema.
- Primary keys or unique constraints can be enforced at the application level, as Iceberg does not natively enforce them.

2.2. SCD Type 2 Logic

- **New Record Insertion**: When a new dimension row is added, the `effective_date` is set to the current date, `expiry_date` is NULL, and `is_current` is `true`.
- **Update to Existing Record**: When a change is detected in a dimension:
    - The existing record is marked as historical by setting its `expiry_date` to the current date and `is_current` to `false`.
    - A new record is created with the updated attribute values, `effective_date` as the current date, `expiry_date`as NULL, and `is_current` as `true`.

3. Partitioning Strategy

Partitioning is essential to optimize queries and reduce the amount of data scanned. Iceberg supports flexible partitioning, including hidden partitioning. For an SCD Type 2 dimensional table, partitioning can be designed to align with how frequently data changes and how queries are typically performed.

3.1. Recommended Partitioning Fields:

- **`is_current`**: Partitioning on the `is_current` column allows you to quickly query active records.
- **`effective_date`**: Partitioning by `effective_date` helps with time-travel queries or filtering based on time ranges.
- **`expiry_date`**: Partitioning by `expiry_date` can be beneficial when analyzing historical records or retrieving past states.

3.2. Example Partitioning Strategy

sql
Copy code
```sql
PARTITIONED BY (is_current, TRUNCYEAR(effective_date))
```

This partitions the table by whether the record is current (`is_current`) and the year of the `effective_date`. This structure reduces scan sizes when looking for current records or querying historical data by year.

3.3. Partition Evolution:

Iceberg allows for **partition evolution**, meaning we can change the partitioning strategy later without rewriting the entire table, which is a powerful feature when your data model changes over time.

4. Best Practices for Storing Data in S3

- **Optimized File Sizes**: Iceberg automatically manages file sizes to ensure optimal performance. Aim for file sizes between 128MB and 1GB. Iceberg's compaction process will merge small files to maintain this size.
- **Snapshot Management**: Iceberg supports **snapshot isolation**, which allows querying of historical data states. Regularly prune old snapshots to avoid excessive storage costs in S3.
- **Metadata Management**: Iceberg stores metadata such as partition layouts, file sizes, and snapshots, which allows for faster query planning. Store this metadata in a well-optimized storage location, separate from the data files if needed.
- **Data Compression**: Use a highly efficient columnar format such as **Parquet** with compression algorithms like **Snappy** or **Zstd** to reduce S3 storage costs and improve read/write performance.

5. Handling SCD Type 2 Data Changes in Iceberg

Iceberg's **merge-on-read** capabilities make it easy to handle data updates, which are essential for SCD Type 2 tables.

5.1. Merge Operations:

When updating a dimension record, the process involves:

1. **Read**: Fetch the existing record from the dimension table using the `natural_key`.
2. **Mark Expired**: Update the existing record by setting the `expiry_date` and marking it as non-current (`is_current = false`).
3. **Insert New Record**: Insert the new record with the updated attributes and set `is_current = true`.

5.2. Example Pseudocode for Handling SCD Type 2 Changes:

python
Copy code
```
from pyspark.sql import SparkSession
from datetime import datetime

# Initialize Spark session and load the Iceberg table
spark = SparkSession.builder.getOrCreate()
```

```
dim_table =
spark.read.format("iceberg").load("path_to_table")

# Example change in a record
def update_dimension_table(natural_key, updated_values):
    current_date = datetime.now().date()

    # Step 1: Fetch the current record
    existing_record =
dim_table.filter((dim_table['natural_key'] == natural_key) &
(dim_table['is_current'] == True))

    # Step 2: Mark the existing record as expired
    if existing_record.count() > 0:
        expired_record =
existing_record.withColumn("expiry_date",
current_date).withColumn("is_current", False)

expired_record.write.format("iceberg").mode("overwrite").save
("path_to_table")

    # Step 3: Insert the new updated record
    new_record = {
        'dimension_id': generate_new_surrogate_key(), # A new
surrogate key
        'natural_key': natural_key,
        'attribute_1': updated_values['attribute_1'],
        'attribute_2': updated_values['attribute_2'],
        'effective_date': current_date,
        'expiry_date': None,
        'is_current': True,
        'created_at': current_date,
        'updated_at': current_date
    }

spark.createDataFrame([new_record]).write.format("iceberg").m
ode("append").save("path_to_table")
```

6. Additional Iceberg Features for SCD Type 2

- **Time Travel**: Iceberg allows users to query the state of the table at any specific point in time. This is especially useful for auditing purposes or reconstructing historical reports.

sql
Copy code
```
SELECT * FROM dim_table.snapshot_at('2023-09-01T12:00:00')
```

- **Schema Evolution**: You can evolve the schema (e.g., add new attributes to your dimension) without rewriting the whole table. This supports flexible changes in the data model.

sql

Copy code

```
ALTER TABLE dim_table ADD COLUMN new_attribute STRING
```

- **Compaction**: Iceberg automatically compacts small files, but periodic manual compaction may be beneficial for improving query performance in very large datasets.

7. Conclusion

This Iceberg table design for dimensional modeling with SCD Type 2 effectively tracks historical data changes, ensures optimal performance on S3, and leverages Iceberg's advanced features such as schema evolution, time travel, and partitioning. This approach provides a robust, scalable solution for handling large datasets and their evolving requirements in a modern data platform.