Guide for Data Quality Management in an infrastructure complex of Hadoop clusters integrated with a Supercomputer

Contents

1. Integration of Big data system HADOOP with a Supercomputer

In NCC Bulgaria, a Big Data System HADOOP managed by UNWE is integrated with Supercomputer located in Tech Park Sofia for the purpose of extensive and complex processing. The exact structure of the Big Data System is consisting of 4 Hadoop clusters – Centralized one located in UNWE with about 4,5 PB disk space and 3 other Hadoop clusters located in Technical university Gabrovo, in Plovdiv university and in University Ruse- fig.1

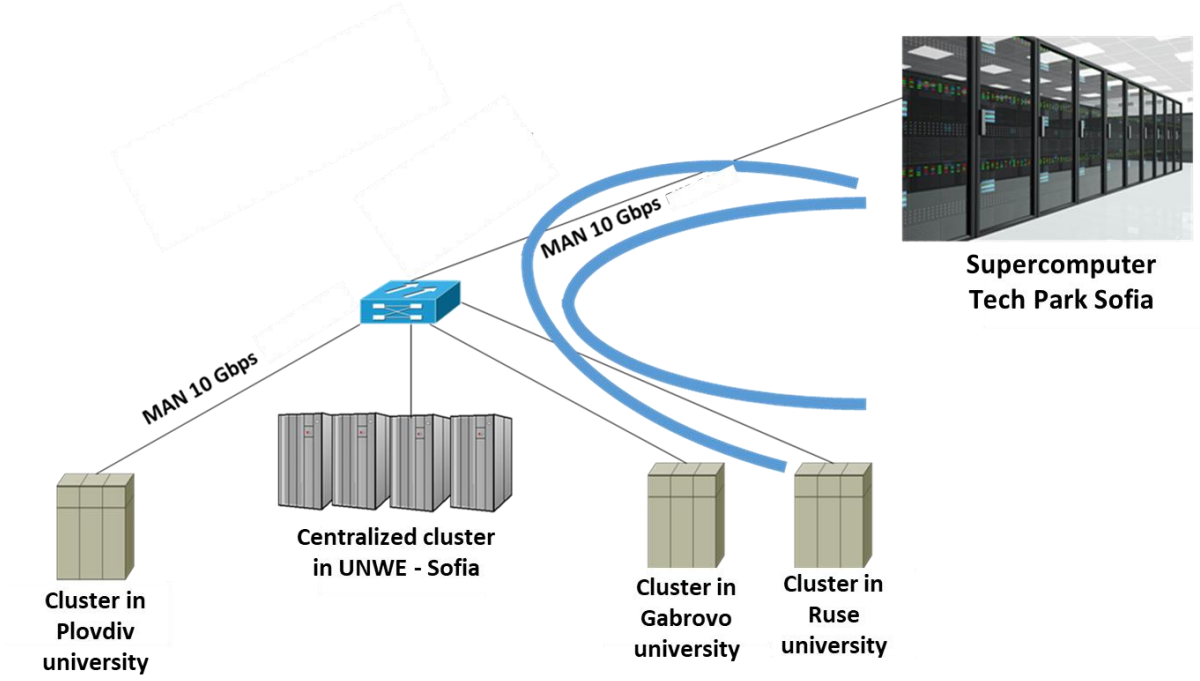

Fig. 1

Generally, the role of the presented infrastructure is to provide the necessary Data security and to provide Data quality. While in this Guide we will not emphasis on Data security, the focus of Data Quality Management is on the functioning concentrated on the Big Data Hadoop clusters (HPDA systems). Data Quality Management is provided on 2 phases: Entrance Data Management and Final Data Management – fig.2.

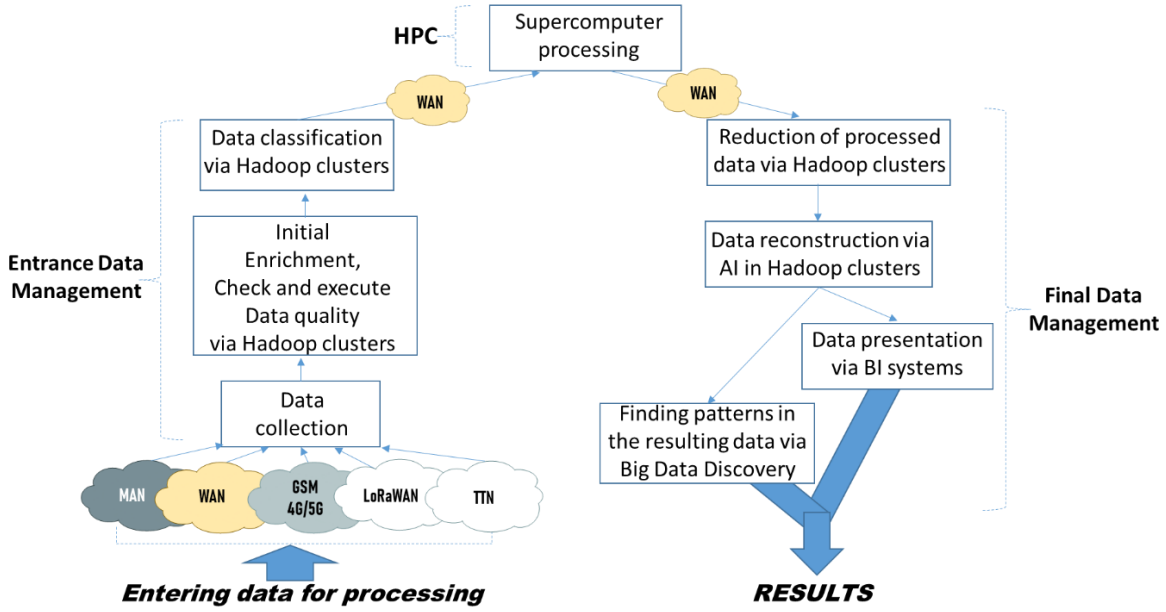

Fig.2

Entrance Data Management provides Data collection using all main networks – MAN, WAN, GSM 4G/5G, LoRaWAN and International LoRaWAN as TTN. The process of collection is organized in 2 ways – storing of data (mainly structured and semi-structured data, as well as some non-structured data as pictures and graphics) and passing true data streams (mainly video and audio). After data is collected, it goes to Initial enrichment and Check and execute Data quality.

When integrating a Big data system as Hadoop system with a Supercomputer, Initial data enrichment plays a crucial role in preparing raw data for high-performance computing workflows. The Initial enrichment covers:

- Extract data from one or more original sources – through the mentioned 5 types of multi-source networks;
- Transform data into the way processing processes require it - grouping, standardizing, sorting, sorting, merging, recoding, creating surrogate keys;
- Load data into respective processing storage - full load, incremental load, default load, overlap load, interval load;
- Provide a unified point of view - provides a unified view of the data, aggregation, formation of information-knowledge from the data;
- Provide historical context - creates a long-term view of data so that older data sets can be viewed alongside newer data, as well as whether the new data matches the content of the old data.

The execution of Data Quality consists of 2 steps – Discovering of the level of the quality and Correction of data to the appropriate level of quality – execution of the needed quality. This covers the following elements of the Data quality:

i. Accuracy - verifiable source
ii. Completeness - provides all necessary values
iii. Consistency - the existence of some data requires the existence of other specified data
iv. Validity - collection is against specified business rules, in a specified format and range
v. Uniqueness - no duplication or overlapping of values

vi. Timeliness - data exists when needed for use

According to MIT, correcting data quality and removing errors from poor-quality data ensures an average of 15%-25% of companies' annual revenue, and at the same time, according to IBM, maintaining Data Quality requires billions of USD per year worldwide. For this reason, the main attention of the current Guide is on Data Quality Management, applying Hadoop clusters as entrance point to the data before treatment of the Supercomputers (High Performance computers).

2. Architecture of Hadoop system supporting Entrance Data Management

The IT Architecture of Hadoop system (one or a few Hadoop clusters) supporting the Entrance Data Management in a complex Hadoop-Supercomputer is presented in figure 3.

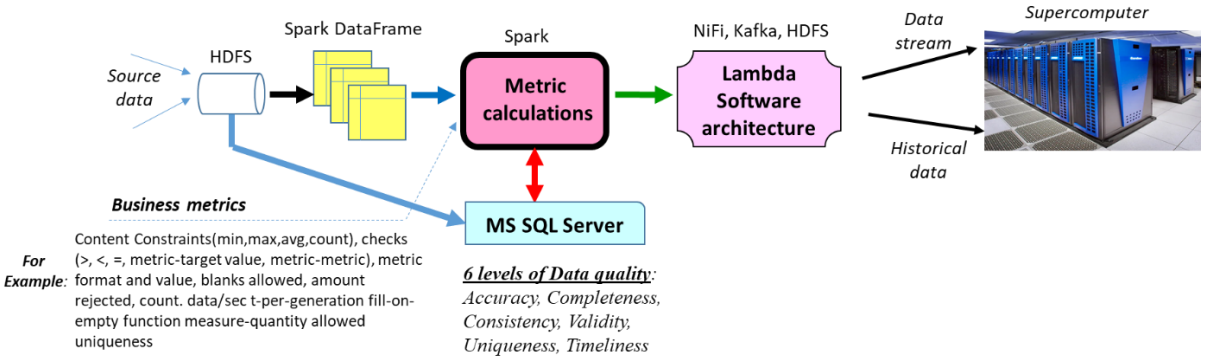

Fig.3

This Architecture was developed and tested in the Scientific Infrastructure of the Centralized Hadoop cluster in UNWE.

The hearth of the approach is using Spark for Hadoop working together with MS SQL Server, where Spark is installed in many as possible Hadoop data nodes. The purpose of such Spark process is to provide metrics for the data and all data to be evaluate according of these metrics. Example of such metrics are: Content Constraints (min, max, average, count), checks (>, <, =, metric-target value, metric-metric), metric format and value, blanks allowed, amount rejected, number of data/sec, t-per-generation, function on filling in empty field, allowed uniqueness. These metrics will be applied in Spark script, receiving as a result for each data item appropriate levels from the 6 Levels of Data quality: Accuracy, Completeness, Consistency, Validity, Uniqueness, Timeliness. There are 2 types of Entrance Data Management – Process based and AI based. Process based data quality management is recommended for structured and semi-structured data, while AI based data quality management is recommended for semi-structured and non-structured data.

The incoming data is stored in HDFS (core Hadoop file system) and from there data is transferred or to Spark DataFrames for treating by Spark, or to MS SQL Server files for treating by MS SQL Server. The IT Architecture for the process of Metrics analysis and enrichment is presented in figure 4.

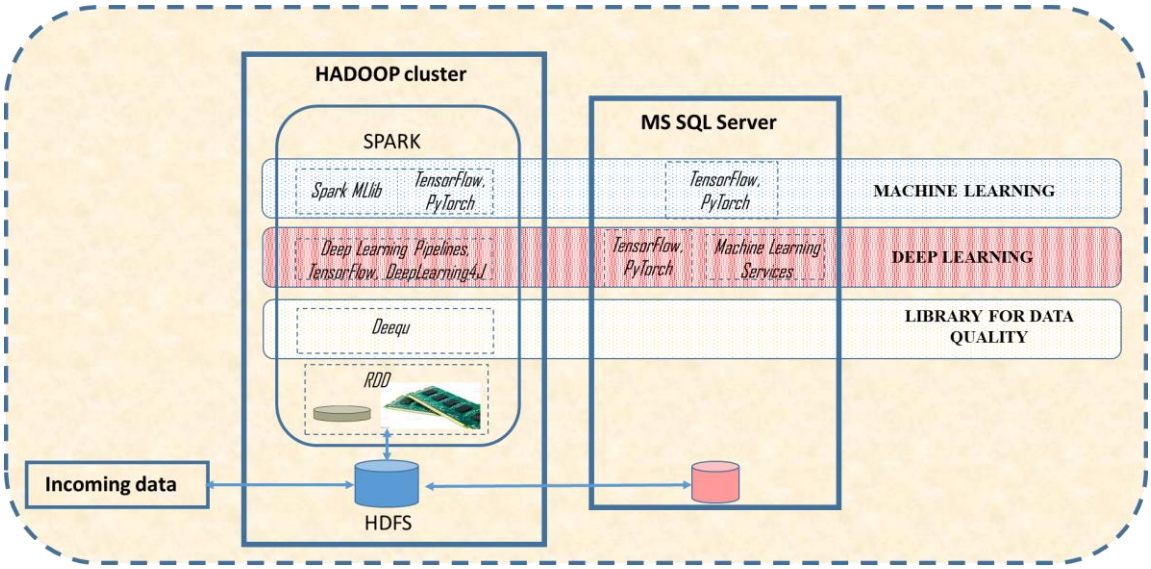

Fig.4

In the figure are presented the recommended tools and Libraries to use for the Metrics calculation and enrichment. It is important to mention that form the processing data quality evaluation and enrichment, the Library Deequ - PyDeequ is used, which works only on Spark. On the figure are mentioned the ways of providing AI based Entrance Data Management, using Machine Learning and/or Deep Learning. The most of the data in the business requiring quality enrichment are structured and semi-structured, and for this we will pay special attention on Procedural based Entrance Data Management, using Deque Library.

Lambda Software Architecture is a data processing design pattern that aims to handle massive quantities of data (Big Data) by combining both batch processing and real-time (stream) processing methods using 2 paths for each processing. It's designed to provide a comprehensive, accurate, and low-latency view of data, addressing the challenges of processing large volumes of rapidly generated information. The Batch path is responsible for processing all historical data. It stores the master dataset (an immutable, append-only record of all data) and pre-computes results (batch views) from this complete dataset. The Real-time path handles real-time data that hasn't yet been processed by the batch layer. It processes data streams as they arrive, providing low-latency, near real-time insights. Due to its focus on speed, the views generated by the Real-time path might be less accurate or complete than those from the batch path. The results from Batch path and Real-time path is integrated into a Service path, integrating data from both paths and making the processed data available for queries to end-users and applications. In our Guide, we recommend using and we have used Apache HBase. In the Lambda Software Architecture New the incoming data is fed into both the Batch path and the Real-time simultaneously. The Batch path processes this data along with all historical data to create accurate, pre-computed views. The Real-time path processes the incoming data in real-time to provide immediate, though potentially less precise, insights. The Service path then combines these two sets of views, allowing users to query all data, including the most recent, with low latency. Once the Batch path has processed the data that the Real-time path was handling, the speed layer's temporary views for that data are superseded. Lambda Software Architecture is well-suited for scenarios requiring both historical analysis and immediate insights from large datasets, like: Social Media Analytics (Sentiment analysis, trend detection), Fraud Detection (Analysing historical transaction patterns to detect anomalies in real-time), IoT Data Analytics (Processing sensor data for predictive maintenance, real-time

monitoring), Personalized Marketing Campaigns (Analysing customer behaviour to deliver real-time recommendations), Real-time Dashboards and Reporting (Providing up-to-date business intelligence).

3. Essence of DEEQU Library

Deequ Library provides a robust and scalable framework for integrating data quality checks directly into your data pipelines, helping to ensure the reliability and trustworthiness of your data.

The essence of the Deequ library can be summarized as **"unit tests for data"** at scale, built on Apache Spark**.**

Here's a breakdown of its key components and purpose:

- **Unit Tests for Data:** Just like a programmer writes unit tests for the code to ensure it behaves as expected, Deequ allows to define "unit tests" for the data. These tests are expressed as **constraints** that specify what constitutes "good" data.
- **Data Quality Measurement:** Deequ measures various aspects of data quality by computing **metrics** on large datasets. These metrics can include:
    - **Completeness:** Are there missing values?
    - **Uniqueness:** Are there duplicate entries where there shouldn't be?
    - **Validity:** Do values conform to expected patterns or ranges (e.g., "high" or "low" for a priority column, non-negative numbers)?
    - **Consistency:** Are relationships between columns as expected (e.g., correlation)?
    - **Schema conformance:** Does the data adhere to its expected structure (column names, types)?
- **Built on Apache Spark:** Deequ leverages the distributed processing power of Apache Spark, making it suitable for analyzing and validating **very large datasets** (billions of rows) efficiently. This is crucial for big data environments.
- **Proactive Error Detection:** The core idea is to find data errors *early* in the data pipeline, before the data is consumed by downstream systems, machine learning models, or used for critical business decisions. By catching issues early, a programmer can "quarantine" and fix bad data, preventing cascading failures or incorrect outputs.
- **Key Functionality:**
    - **Metrics Computation:** Automatically calculates various data quality statistics (e.g., min, max, mean, completeness, distinctness, correlation).
    - **Constraint Verification:** Allows users to define custom data quality rules (constraints) and then verifies if the data satisfies these rules. It generates a report indicating successes and failures.

- **Constraint Suggestion:** Can automatically profile data and suggest a set of reasonable constraints based on the observed data patterns, helping users get started quickly.
- **Metrics Repository:** Provides mechanisms to persist and track data quality metrics over time, enabling monitoring and anomaly detection.
- **Anomaly Detection:** Can be used to detect unusual changes in data quality metrics over time, signaling potential data issues.

In Deequ Library there are **Methods** for assessing data quality through specific quality parameters, where one method is used for one quality parameter. The set of methods are:

- ApproxCountDistinct
- ApproxQuantile
- ApproxQuantiles
- Completeness
- Compliance
- Correlation
- CountDistinct
- DataType
- Distinctness
- Entropy
- Maximum
- Mean
- Minimum
- MutualInformation
- PatternMatch
- Size
- Sum
- UniqueValueRatio
- Uniqueness

Each method accepts external data and generate as a result - the corresponding estimate via the parameter.

Before we define checks on the data, we want to calculate some statistics on the dataset; we call them metrics. Deequ supports the following **metrics** – Table 1.

Table 1

Metric	Description	Usage Example
ApproxCountDistinct	Approximate number of distinct value, computed with HyperLogLogPlusPlus sketches.	ApproxCountDistinct("review_id")
ApproxQuantile	Approximate quantile of a distribution.	ApproxQuantile("star_rating", quantile = 0.5)
ApproxQuantiles	Approximate quantiles of a distribution.	ApproxQuantiles("star_rating", quantiles = Seq(0.1, 0.5, 0.9))
Completeness	Fraction of non-null values in a column.	Completeness("review_id")
Compliance	Fraction of rows that comply with the given column constraint.	Compliance("top star_rating", "star_rating >= 4.0")
Correlation	Pearson correlation coefficient, measures the linear correlation between two columns. The result is in the range [-1, 1], where 1 means positive linear correlation, -1 means negative linear correlation, and 0 means no correlation.	Correlation("total_votes", "star_rating")
CountDistinct	Number of distinct values.	CountDistinct("review_id")
DataType	Distribution of data types such as Boolean,	DataType("year")

	Fractional, Integral, and String. The resulting histogram allows filtering by relative or absolute fractions.	
Distinctness	Fraction of distinct values of a column over the number of all values of a column. Distinct values occur at least once. Example: [a, a, b] contains two distinct values a and b, so distinctness is 2/3.	Distinctness("review_id")
Entropy	Entropy is a measure of the level of information contained in an event (value in a column) when considering all possible events (values in a column). It is measured in nats (natural units of information). Entropy is estimated using observed value counts as the negative sum of (value_count/total_count) * log(value_count/total_count). Example: [a, b, b, c, c] has three distinct values with counts [1, 2, 2]. Entropy is then (-1/5*log(1/5)-2/5*log(2/5)-2/5*log(2/5)) = 1.055.	Entropy("star_rating")
Maximum	Maximum value.	Maximum("star_rating")
Mean	Mean value; null values are excluded.	Mean("star_rating")

Minimum	Minimum value.	Minimum("star_rating")
MutualInformation	Mutual information describes how much information about one column (one random variable) can be inferred from another column (another random variable). If the two columns are independent, mutual information is zero. If one column is a function of the other column, mutual information is the entropy of the column. Mutual information is symmetric and nonnegative.	MutualInformation(Seq("total_votes", "star_rating"))
PatternMatch	Fraction of rows that comply with a given regular experssion.	PatternMatch("marketplace", pattern = raw"\w{2}".r)
Size	Number of rows in a DataFrame.	Size()
Sum	Sum of all values of a column.	Sum("total_votes")
UniqueValueRatio	Fraction of unique values over the number of all distinct values of a column. Unique values occur exactly once; distinct values occur at least once. Example: [a, a, b] contains one unique value b, and two distinct values a and b, so the unique value ratio is 1/2.	UniqueValueRatio("star_rating")

Uniqueness	Fraction of unique values over the number of all values of a column. Unique values occur exactly once. Example: [a, a, b] contains one unique value b, so uniqueness is 1/3.	Uniqueness("star_rating")

4. Main operations for Data Quality Management with Deequ Library

The main operations for data quality management with Deequ library can be categorized into these core components:

4.1. Metrics Computation:

- Deequ calculates various data quality metrics (statistics) on a dataset. These metrics provide insights into the properties of your data.
- Examples of metrics include:
  - `Size()`: The total number of rows.
  - `Completeness("column_name")`: The percentage of non-null values in a column.
  - `Uniqueness("column_name")`: The percentage of unique values in a column.
  - `Distinctness("column_name")`: The ratio of distinct values to the total number of values.
  - `Minimum("column_name")`, `Maximum("column_name")`, `Mean("column_name")`, `Sum("column_name")`, `StandardDeviation("column_name")`: Statistical measures for numerical columns.
  - `Correlation("column1", "column2")`: Measures the correlation between two numerical columns.
  - `Compliance("column_name", "expression")`: Checks if a certain expression holds true for a column.
- Deequ leverages Spark's distributed processing capabilities to efficiently compute these metrics on large datasets.

4.2. Constraint Verification:

- This is where the designer can define its data quality expectations as a set of constraints. Deequ then verifies if the data adheres to these predefined constraints.
- It is possible to define checks with a `Check` object, specifying a `CheckLevel` (e.g., `Error`, `Warning`) and then add various constraints to it.

- Examples of constraints that can be defined:
  - `isUnique("column_name")`: Ensures all values in a column are unique.
  - `hasSize(_ >= min_size)`: Verifies the dataset has at least a certain number of rows.
  - `isNotNull("column_name")`: Checks for the absence of null values.
  - `hasDataType("column_name", DataType.StringType)`: Verifies the data type of a column.
  - `hasMin("column_name", min_value)`: Checks if the minimum value in a column meets a threshold.
  - `isContainedIn("column_name", Seq("value1", "value2"))`: Ensures column values are from a predefined set.
  - `satisfies("column_name", "column_name > 0")`: Allows defining custom SQL-like expressions for validation.
- Deequ generates a data quality report (`VerificationResult`) indicating whether each constraint passed or failed. This report can be converted to a DataFrame for easy analysis.

4.3. Constraint Suggestion:

- If it is unsure what constraints to define, Deequ can help by automatically suggesting potential data quality constraints based on profiling your data.
- The `ConstraintSuggestionRunner` profiles the data, infers patterns, and proposes a set of meaningful constraints that you can then review and incorporate into your verification suite. This is particularly useful for exploring new datasets or when the designers don't have explicit data quality requirements yet.

4.4. Anomaly Detection:

- Deequ can detect anomalies in data quality metrics over time. Instead of fixed thresholds, it can learn the normal behaviour of metrics and flag deviations as anomalies. This is crucial for continuous data quality monitoring in evolving data pipelines.

4.5. Use Metrics Repository (Persistence and Querying):

- Deequ provides a `MetricsRepository` interface to store and load computed metrics. This allows the programmer to persist its data quality check results for historical analysis, trend monitoring, and auditing.
- The programmer can save results to various systems, such as file systems (e.g., JSON), databases (e.g., InfluxDB with custom adapters), or in-memory.

4.6. Incremental Metrics Computation:

- For large, partitioned datasets, Deequ supports incremental metrics computation. This means it can efficiently update metrics for new or changed partitions without re-reading the entire dataset, saving significant computational resources and time.

5. Installation steps of Amazon Deequ on Hadoop Spark

Installing Amazon Deequ on Hadoop Spark involves adding the Deequ library as a dependency to the project. The specific steps depend on programmer's build system (Maven or SBT) and Spark version. Here's a breakdown for both scenarios:

Using Maven:

1. **Check Spark Version:** Ensure you know your exact Apache Spark version. Deequ has different versions compatible with various Spark releases.
2. **Add Deequ Dependency:** In your project's pom.xml file, add the following dependency section, replacing <version> with the appropriate Deequ version based on your Spark version.

For Spark 3.1.x:

XML

```
<dependency>
  <groupId>com.amazon.deequ</groupId>
  <artifactId>deequ</artifactId>
  <version>2.0.0-spark-3.1</version>
</dependency>
```

Using SBT:

1. **Check Spark Version (same as Maven):** Identify your Spark version.
2. **Add Deequ Dependency:** In progarmmer's build.sbt file, add the following line, again replacing <version> with the compatible Deequ version for your Spark.

For Spark 3.1.x:

```
libraryDependencies += "com.amazon.deequ" % "deequ" % "2.0.0-spark-3.1"
```

Important points to remember:

- Refer to the Deequ documentation for the latest version information and compatible Spark versions: https://github.com/awslabs/deequ
- If the programmer is using an older Spark version (below 3.1), he willl need to use a Deequ 1.x version (consult Deequ documentation for details).
- Make sure the program system has Java 8 installed, as Deequ depends on it.

By following these steps and considering the compatibility aspects, the programmer should be able to successfully install Amazon Deequ on the Hadoop Spark environment.

6. Using data for testing of Amazon Deequ

Data selection for testing Amazon Deequ depends on the specific aspects the designer wants to evaluate. Here are some approaches to consider:

1. **Sample Data:**
    - The designer can leverage sample datasets from Deequ's documentation or GitHub repository https://github.com/awslabs/deequ. These samples showcase basic usage and cover various data types.
    - If the designer has an existing small-scale dataset mirroring your production data structure, that can also work for initial testing Deequ's functionalities.
2. **Synthetic Data Generation:**
    - Tools like Apache Spark's sql.functions or libraries like ScalaTest provide functions to generate synthetic data with specified characteristics. This allows the designer to test Deequ's behavior on data with controlled distributions and outliers.
3. **Real-world Data Snippets:**
    - Extract a limited portion of anonymized data from the production environment. Ensure it represents the data structure and potential quality issues you expect in the actual use case.

Remember, Deequ is built for Apache Spark and works on tabular data. So, the designer's chosen data should be transformable into a Spark DataFrame.

Here are some additional tips for selecting data for Deequ testing:

- **Focus on data types in use:** If the data contains specific data types (e.g., dates, geolocations), include them in the test data to ensure Deequ handles them correctly.
- **Simulate potential issues:** Introduce controlled errors or missing values in the test data to verify Deequ identifies these quality problems.
- **Consider data size:** While Deequ is designed for large datasets, start with a manageable size for initial testing before scaling up.

7. Conversion of HDFS data to a Spark DataFrame

To convert HDFS data to a Spark DataFrame, the programmer can use the spark.read method with the appropriate file format and HDFS path. Here are a few examples:

7.1. Reading a CSV file from HDFS:

```
booksSchema = StructType() \
      .add("id", "integer") \
      .add("book_title", "string") \
      .add("publish_or_not", "string") \
      .add("technology", "string")
booksdata = spark.read.csv("hdfs://localhost:9000/dezyre_books", schema=booksSchema)
booksdata.show(5)
```

This code snippet reads a CSV file from the HDFS path hdfs://localhost:9000/dezyre_books and assigns the specified schema to the DataFrame .

7.2. Reading a text file from HDFS:

```
df = spark.read.text("hdfs://nn1home:8020/text01.txt")
```

This code snippet reads a text file from the HDFS path hdfs://nn1home:8020/text01.txt and creates a DataFrame with a single column named "value".

7.3. Reading a JSON file from HDFS:

```
df = spark.read.json("hdfs://nn1home:8020/file.json")
```

This code snippet reads a JSON file from the HDFS path hdfs://nn1home:8020/file.json and creates a DataFrame with a schema inferred from the input file.