POLYTECHNIC OF TURIN

Master's Degree in Computer Engineering

Master's Degree Thesis

Extension and improvement of Data Quality and Observability framework through DBT

Supervisor

Prof. Paolo GARZA

Candidate

Giorgio CACOPARDI

April 2025

Abstract

The growing complexity and volume of data in modern business intelligence architectures requires increasing attention to data quality and traceability. In this context, improved data monitoring and validation processes are essential to ensure the reliability and correctness of analyses.
This thesis explores the extension of Alertable, an existing platform focused on data quality and observability, by integrating it with DBT (Data Build Tool), one of the most popular tools for data management and transformation within data engineering pipelines.

The main objective of this work is to develop new functionalities that improve Alertable's ability to monitor and validate data through the integration of advanced data quality tests, allowing anomalies and discrepancies in data flows to be automatically detected. The proposed extension involves the creation of a framework to perform quality tests directly in DBT or eventually convert DBT tests into the format used by Alertable, leveraging Alertable's data validation capabilities.

Acknowledgements

"Computers are incredibly fast, accurate, and stupid; humans are incredibly slow, inaccurate, and brilliant; together they are powerful beyond imagination."

Albert Einstein

Table of Contents

Chapter 1

Introduction

In recent years, the importance of data management in business applications has grown exponentially, transforming the way organizations leverage information to improve processes and decisions. Data quality and reliability have become strategic priorities, with the emergence of tools and frameworks to ensure that data are not only correct, but also constantly monitored.

The ever-increasing need to rely on data-driven methodologies that enable more reliable business choices or the use of generative AI tools require increasingly complex IT infrastructures capable of being able to handle the sheer volume of them.

1.1 Motivation

In this context, this thesis work focuses on the integration of DBT (Data Build Tool) within Alertable, a platform that addresses data quality, observability and profiling. Alertable aims to provide an ecosystem that monitors data flows, identifies anomalies, and notifies stakeholders in real time. DBT, on the other hand, is a framework for managing, transforming, and orchestrating data with a modular structure, improving the traceability and maintainability of data pipelines. The join of this 2 tools can be a unifying point to allow figures who are less technical but more focused on business knowledge and the usefulness in that context of data to be able to set up data control flows in a more understandable way.

1.2 Purpose and goal

The goal is to integrate the DBT tool within the Alertable flow, allowing the execution of data quality jobs that can perform checks on one or more different datasources, the use of external DBT libraries for data quality tests to extend the type of tests possible and the conversion of DBT tests to the format currently used by Alertable to ensure interchangeable use of the two modes.

1.3 Methodology

The adopted methodology is made up of different facts:

1. At first, the DBT tool was studied, in order to acknowledge its configuration and use
2. Secondly, an attempt was made to understand which components were the most useful in order to integrate it into Alertable
3. Goals have been set about the functionality to be achieved:
   - Creating a DBT project that can handle multiple different datasources, ranging from connections with Athena, My Sql or Postgres
   - Ability to automatically import from a YAML file of checks directly to Alertable
   - Execution of jobs that can simultaneously run multiple DBT checks on multiple different datasources, resulting in retrieval of the outcome and failed rows for every single check
   - Conversion of DBT check to the default Alertable Check

1.4 Outline

- Chapter 2 introduces the concept of Data Quality, Observability and Profiling, the components typically required to achieve them, and the state of the art technology landscape in this regard
- Chapter 3 provides an explanation and overview of the Alertable application, showing its functionality and how to use it
- Chapter 4 describes DBT, all of its components, the typical structure of a DBT project, and its functionality
- Chapter 5 covers the work done about the integration of DBT into Alertable, showing the technologies used, the various goals and functionality achieved, and a demonstration of their use
- Chapter 6 covers a comparison about efficiency and scalability between the job execution with the AWS EMR cluster adopted by Alertable and the DBT Job execution

Chapter 2

State of the Art

2.1 Introduction to Data Quality, Observability and Profiling

Data Quality

"*Data quality measures how well a dataset meets criteria for accuracy, completeness, validity, consistency, uniqueness, timeliness and fitness for purpose, and it is critical to all data governance initiatives within an organization.*" [1].

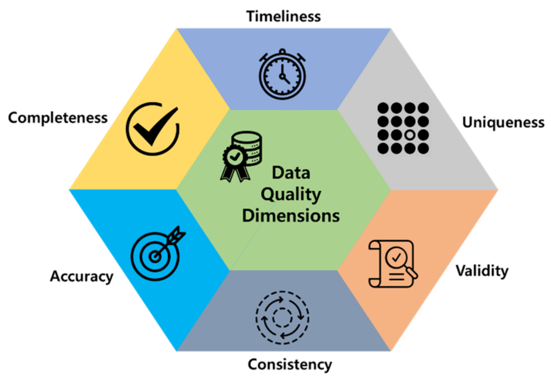


Figure 2.1: Data quality definition

- **Accuracy:** it refers to how well the data correctly represents real-world values and its trustworthy.
- **Completeness:** it evaluates whether all required data is present and available.
- **Consistency:** it examines whether data is consistent across different sources or systems.

- **Timeliness:** it measures whether the data is up-to-date and available, when needed.
- **Validity:** it refers to whether the data conforms to the defined business rules, formats or constraints.
- **Uniqueness:** it ensures that the data does not have unnecessary duplicates.

Data quality can be affected at any stage of the data pipeline - before ingestion, in production or even during analysis. Maintaining high data quality is not just critical for everyday business operations: it becomes even more essential as companies adopt Artificial Intelligence (AI) and automation technologies. Over the past few years, ever more companies have tried to adopt the DevOps principle (shortening the systems development life cycle by applying practices like Continuous integration/Continuous deployment and micro services architecture) to data in the form of "DataOps", by automatize all the steps involved in building data pipelines, managing data workflows, and ensuring the reliability, quality, and security of data processes.

Data Observability

Data Observability refers to the ability to monitor, track and understand the health, quality and performance of data systems throughout their life cycle. It involves capturing metrics, logs, and events to provide real-time insights into the behavior of data processes. The five pillars of data observability are [2]:

- **Freshness:** it seeks to understand how up to date the data in the tables are, as well as the cadence at which they are updated
- **Quality:** it refers to aspects about data itself like percent NULLS, percent Uniques or if the data is within an accepted range.
- **Volume:** it refers to the completeness of the data in the tables and offers insights on the health of your data sources
- **Schema:** it refers to the monitoring about changes in the organization of the data
- **Lineage:** it provides the answer on which part of the data pipeline were affected, as well as which teams are generating the data and who is accessing to it

Data Profiling

Data profiling is the process of analyzing, reviewing and summarizing datasets in order to better understand their structure, content, quality and interrelationships. It involves examining data for patterns, inconsistencies, errors and anomalies to ensure it meets the requirements for its intended use. For this reason, it is often a preliminary step in data quality management, data governance or data preparation workflows.

2.1.1 Evolution of Data Quality Challenges

In recent years, there has been a growing and significant focus on the importance of data and digitalization in the fields of business management and decision-making. This trend has led to massive investments in this sector, exponentially increasing the volume of available data and complicating its management. This shift has manifested in various aspects, such as:

- **Migration to cloud platforms:** nowadays always more company base their approach on the cloud due to its easier management, scalability and cost optimization, leasing in the increasing adoption of solutions like Amazon Redshift, Snowflake or Google BigQuery

- **Proliferation of data sources:** companies use from dozens to hundreds of internal and external data sources to produce analytics and ML models.

- **Rising complexity of data pipelines:** data pipelines have become increasingly complex with multiple stages of processing and nontrivial dependencies between various data assets. Without visibility into these dependencies, however, any change made to one data set can have unintended consequences impacting the correctness of dependent data assets

- **The emergency of increasingly specialized and decentralized data teams:** companies increasingly rely on data to drive smart decision making, they are hiring more and more data analysts, data scientists, and data engineers to build and maintain the data pipelines, analytics, and ML models that power their services and products, as well as their business operations

- **A paradigm shift in database management** Much in the same way that software engineering teams transitioned from monolithic applications to microservice architectures, the data mesh is, in many ways, the data platform version of microservices

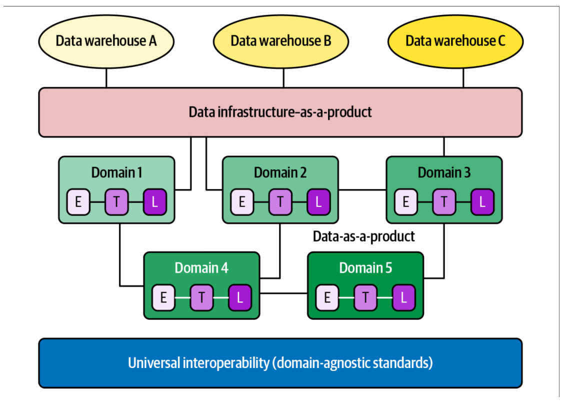


Figure 2.2: Data mesh

2.2 Data Quality in the Modern Data Stack

2.2.1 Core Components

The modern data stack introduces several key components crucial for maintaining data quality.

Data Source Layer

The data source layer is the foundational starting point of any modern data stack, encompassing all systems and mechanisms responsible for generating and storing raw data that will later be processed, analyzed, and utilized for decision-making. This layer serves as the initial interface between the organization's data ecosystem and the external or internal sources from which the data originates.

Data sources can take multiple forms, depending on the type of information being gathered and its intended use. Operational databases are one of the most common types, consisting of transactional systems like relational databases (e.g., PostgreSQL, MySQL, Oracle DB) that store structured, real-time data essential for running daily business operations. These databases are optimized for quick and efficient retrieval and updating of information, forming the backbone of many operational workflows and usually possess a good level od data quality due to their structured nature.

Another significant category within the data source layer is Software-as-a-Service (SaaS) applications, which are cloud-based solutions used for specific business purposes, such as customer relationship management (CRM) tools (e.g., Salesforce), enterprise resource planning (ERP) platforms, and marketing automation tools. SaaS platforms often produce highly granular, event-driven data, capturing everything from user interactions to campaign performance metrics.

In addition, event streams play a critical role in capturing time-sensitive, high-speed data from systems and devices. These streams, often handled through message brokers or event-processing systems like Apache Kafka or AWS Kinesis, are designed to process and transmit data generated in real-time by applications, IoT devices, or web services. They are particularly important in scenarios where immediate insights are required, such as monitoring user activity on a website, tracking the status of IoT devices, or detecting fraud in financial transactions. This data expecially requires greater attention due to its nature and structure which is unpredictable, with possible missing or duplicate values.

The diversity of data sources in this layer reflects the growing complexity of modern data ecosystems. These sources may vary significantly in structure, ranging from highly organized, structured formats in relational databases to semi-structured or unstructured formats in JSON logs, XML files, or raw text. Moreover, integrating these disparate data sources into a unified data pipeline often requires the use of specialized tools and techniques to standardize, clean, and transform the incoming data for further processing.

Data Ingestion Layer

The Data Ingestion Layer is a crucial intermediary step in the modern data stack, responsible for transporting data from diverse source systems into centralized storage solutions, such as data warehouses, data lakes, or hybrid storage systems. This process ensures that raw data is made available for subsequent transformations, analysis, and consumption. The data ingestion layer handles the complexities of extracting data from heterogeneous systems, standardizing it, and loading it into storage in an efficient and scalable manner. There are typically two primary approaches to data ingestion, each suited for different types of use cases: batch ingestion and real-time (or stream) ingestion:

- **Batch ingestion**: Batch ingestion involves collecting data over a specific period and then transferring it in bulk to the target system. This method is ideal for scenarios where data updates are not time-sensitive or where processing efficiency is prioritized over immediacy.

- **Real time ingestion**: Real-time ingestion, on the other hand, captures and transfers data continuously as it is generated. This approach is essential for applications requiring up-to-the-second updates, such as fraud detection, live dashboarding, or monitoring IoT device activity. It usually requires more sophisticated infrastructure to handle the continuous flow of data.

Data Storage Layer

The Data Storage Layer is a critical component of the modern data stack, serving as the central repository where data is securely stored, organized, and prepared for subsequent transformation, analysis, and retrieval. Modern storage systems are designed with three fundamental goals: scalability, to accommodate growing volumes of data; queriability, to ensure efficient access to data for diverse use cases; and cost efficiency, to optimize resource utilization while minimizing storage expenses. Two of the most commonly used types of storage systems in this context are data warehouses and data lakes, each serving distinct purposes and requirements.

- **Data warehouse**: It is a structured storage system designed to integrate, organize, and analyze large volumes of data from multiple sources. Its primary goal is to support business intelligence, reporting, and data analytics functions by providing a single, reliable source of truth for an organization. They are optimized for structured data, such as relational database tables, and excel in handling queries that involve aggregations, filtering, and summarization.
- **Data lake**: It is a more flexible storage solution that can accommodate data in its raw, unstructured, semi-structured, or structured form. Data lakes are particularly well-suited for big data use cases, such as machine learning and advanced analytics, where diverse data types and formats must coexist. Unlike data warehouses, which enforce a schema-on-write approach (where data is structured during ingestion), data lakes adopt a schema-on-read approach, allowing for greater adaptability

Transformation Layer

In this layer are applied all the business oriented transformations with the goal of making the data a solid source of transformations with different applications, from business intelligence to machine learning model input. Usually, in this step are used tools which can manage a massive amount of data, like Spark, which exploits distributed computing in a cluster to divide the jobs and reduce the latency. There are different tools and ways to apply the transformations like:

- **Distributed Computing Systems**: like Apache Spark or Databricks, where the tasks are split across a cluster to process massive datasets in parallel
- **Cloud based ETL platforms**: usually tools like DBT, for modular SQL based transformations, or AWS Glue for cloud native ones.
- **Relational and Analytical Databases**: they usually leveraging database specific transformations capabilities in data warehouses like Snowflake or Redshift

Quality Control Layer

The role of the data quality layer is to apply different kind of tests to the data to ensure their reliability, robustness and solidity. Data quality issues, if left unchecked, can lead to inaccurate insights, flawed decisions, and operational inefficiencies. Therefore, the quality control layer employs a series of tests, validations, and monitoring mechanisms to proactively identify, address, and prevent data-related issues.

2.3 State of the Art Solutions for Data Quality

This section introduces 2 of the most frequently used tools in the field of data quality.

2.3.1 Great Expectations

Great Expectations is an open-source framework designed to validate, document, and profile data with a focus on ensuring quality and reliability across data workflows. It provides a systematic approach to defining, testing, and enforcing data expectations and it has several major features including data validation, profiling, and documenting of a project.[3]

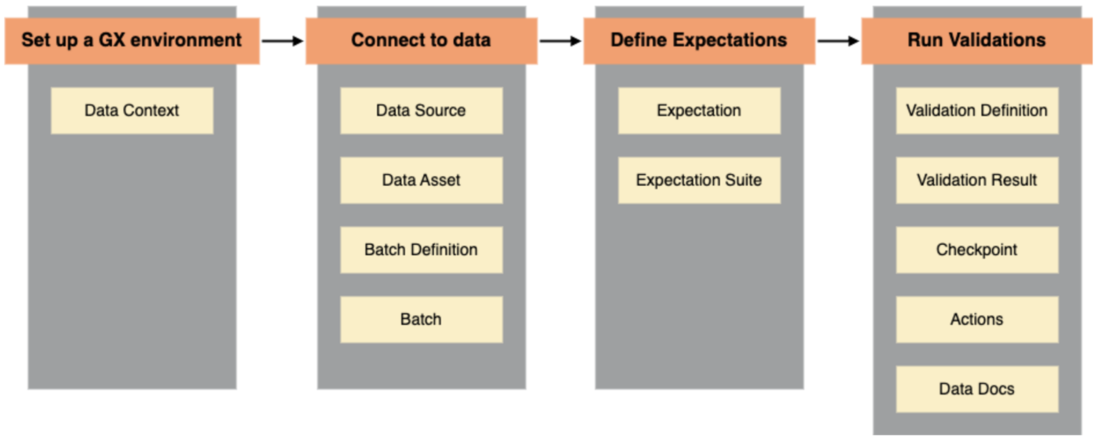


Figure 2.3: Great Expectations overview

Expectations

The expectations represents all the possible types of data quality check that can be performed on a data source. There are a lot of different assertions that can be made like:

Listing 2.1: Example of GE Assertions

```

df.expect_column_values_to_be_unique(column="id")
df.expect_column_values_to_match_regex(
    column="email", regex=r"[^@]+@[^@]+\.[^@]+"
)
df.expect_column_values_to_be_between(
    column="price", min_value=0, max_value=1000
)
df.expect_column_values_to_not_be_null(column="name")
df.expect_column_values_to_be_in_set(
    column="status", value_set=["active", "inactive", "pending"]
)
df.expect_column_value_lengths_to_be_between(
    column="username", min_value=3, max_value=50
)
df.expect_table_row_count_to_be_between(min_value=100, max_value
    =1000)
```

After their definition they can be tested and then it can be obtained the Validation results of them, which constitutes a summary of which and how much data passed the validation[4]:

Listing 2.2: Example of GE Validation

```

{
  "success": false,
  "expectation_config": {
    "expectation_type": "expect_column_max_to_be_between",
    "kwargs": {
      "batch_id": "2018-06_taxi",
      "column": "passenger_count",
      "min_value": 4.0,
      "max_value": 5.0
    },
    "meta": {},
    "id": "38368501-4599-433a-8c6a-28f5088a4d4a"
  },
  "result": {
    "observed_value": 6
  },
  "meta": {},
  "exception_info": {
    "raised_exception": false,
    "exception_traceback": null,
    "exception_message": null
  }
}
```

Expectation Suite

Great Expectations allows the creation of a group of Expectations describing the same set of data by combining all the Expectations defined and evaluate them as a group rather than individually

Listing 2.3: Example of Expectation Suite

```

# Create an Expectation Suite
suite_name = "my_expectation_suite"
suite = gx.ExpectationSuite(name=suite_name)

# Add the Expectation Suite to the Data Context
suite = context.suites.add(suite)

# Create an Expectation to put into an Expectation Suite
expectation = gx.expectations.ExpectColumnValuesToNotBeNull(column="
    passenger_count")

# Add the previously created Expectation to the Expectation Suite
suite.add_expectation(expectation)

# Add another Expectation to the Expectation Suite.
suite.add_expectation(
     gx.expectations.ExpectColumnValuesToNotBeNull(column="
    pickup_datetime")
)
```

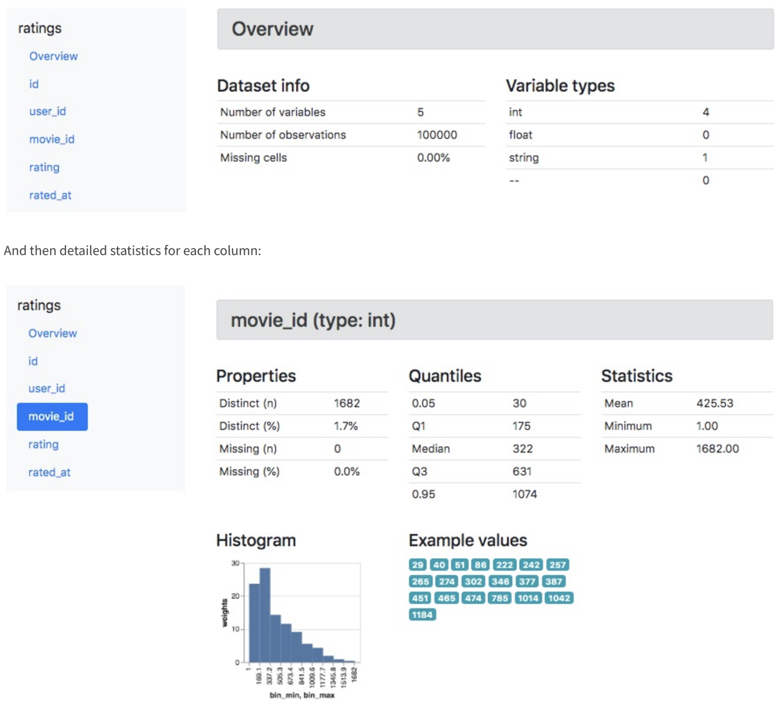
Data docs

It is a feature that consists in the automation of conversion of Expectations, Validation Results, and other metadata into human-readable documentation.2.4

Various data sources

Great Expectations supports the following SQL dialects:

- PostgreSQL
- SQLite
- Snowflake
- Databricks SQL
- BigQuery SQL



Overview

Dataset info

Number of variables	5
Number of observations	100000
Missing cells	0.00%

Variable types

int	4
float	0
string	1
--	0

And then detailed statistics for each column:

movie_id (type: int)

Properties

Distinct (n)	1682
Distinct (%)	1.7%
Missing (n)	0
Missing (%)	0.0%

Quantiles

0.05	30
Q1	175
Median	322
Q3	631
0.95	1074

Statistics

Mean	425.53
Minimum	1.00
Maximum	1682.00

Histogram

Example values

29 40 51 86 222 242 257
265 274 302 346 377 387
451 465 474 785 1014 1042
1184

Figure 2.4: Great Expectations Data Docs[5]

Batch retrieval

Expectations can be individually validated by using a Batch of data, allowing to test the newly created ones or to further understand the data[6]

Listing 2.4: Example of Batch Definition

```

# Retrieve the Batch Definition:
data_source_name = "my_data_source"
data_asset_name = "my_data_asset"
batch_definition_name = "my_batch_definition"
batch_definition = (
    context.data_sources.get(data_source_name)
    .get_asset(data_asset_name)
    .get_batch_definition(batch_definition_name)
)

# Retrieve the first valid Batch of data:
batch = batch_definition.get_batch()

# Or use a Batch Parameter dictionary to specify a Batch to retrieve
# These are sample Batch Parameter dictionaries:

# If you're using File Data Assets, pass values as strings
yearly_batch_parameters = {"year": "2019"}
monthly_batch_parameters = {"year": "2019", "month": "01"}
daily_batch_parameters = {"year": "2019", "month": "01", "day": "01"}

# Otherwise, pass values as integers
integer_daily_batch_parameters = {"year": 2019, "month": 1, "day": 1}


# This code retrieves the Batch from a monthly Batch Definition:

batch = batch_definition.get_batch(batch_parameters={"year": "2019",
    "month": "01"})
```

2.3.2 Soda Core

Soda Core is an open-source data quality and observability tool designed to detect and address data issues throughout the data life cycle. It allows to define checks and validations to monitor data quality.

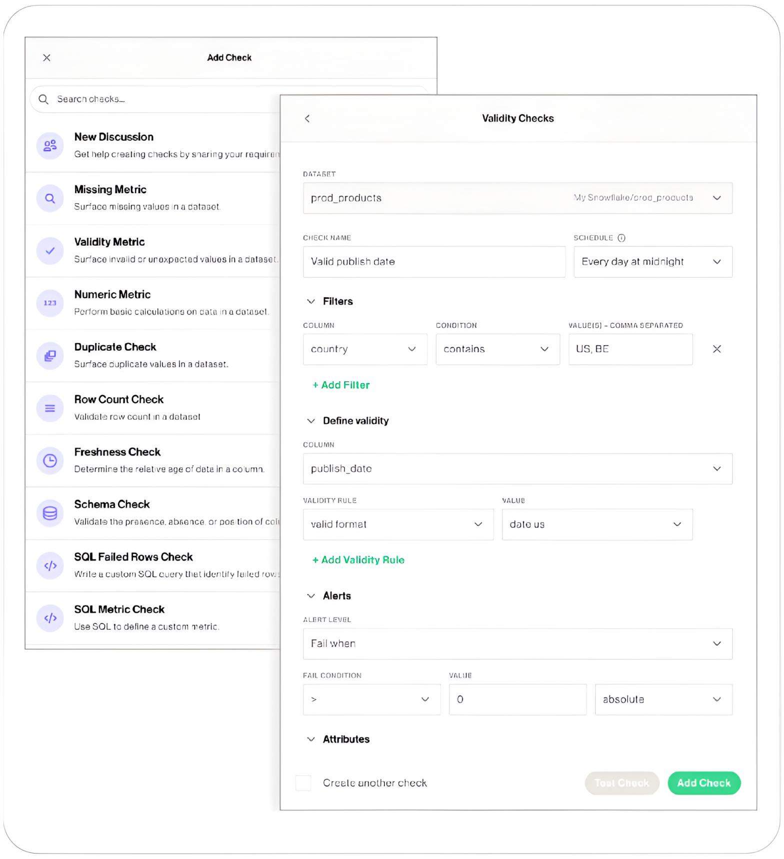
Check and validation

Soda Core has the characteristic of define in a YAML-based domain specific language or programmatic invocation dome data assertions and to return a summary of the results. Here an example of a possible configuration[7]:

Listing 2.5: Example of Soda Core Checks

```

# Checks for basic validations
checks for dim_customer:
  - row_count between 10 and 1000
  - missing_count(birth_date) = 0
  - invalid_percent(phone) < 1 %:
      valid format: phone number
  - invalid_count(number_cars_owned) = 0:
      valid min: 1
      valid max: 6
  - duplicate_count(phone) = 0

checks for dim_product:
  - avg(safety_stock_level) > 50
# Checks for schema changes
  - schema:
      name: Find forbidden, missing, or wrong type
      warn:
        when required column missing: [dealer_price, list_price]
        when forbidden column present: [credit_card]
        when wrong column type:
          standard_cost: money
      fail:
        when forbidden column present: [pii*]
        when wrong column index:
          model_name: 22

# Check for freshness
  - freshness (start_date) < 1d

# Check for referential integrity
checks for dim_department_group:
  - values in (department_group_name) must exist in dim_employee (
    department_name)
```

Soda also has the possibility, by using the paid version Soda Cloud, to define checks by using a no code GUI, where it can be also set a schedule and defined some specific condition on the check results to trigger an alert


Add Check
Search checks...
New Discussion
Get help creating checks by sharing your requirem
Missing Metric
Surface missing values in a dataset.
Validity Metric
Surface invalid or unexpected values in a dataset.
Numeric Metric
Perform basic calculations on data in a dataset.
Duplicate Check
Surface duplicate values in a dataset.
Row Count Check
Validate row count in a dataset
Freshness Check
Determine the relative age of data in a column.
Schema Check
Validate the presence, absence, or position of col
SQL Failed Rows Check
Write a custom SQL query that identify failed rows
SQL Metric Check
Use SQL to define a custom metric.
Validity Checks
Filters
+ Add Filter
Define validity
+ Add Validity Rule
Alerts
Attributes
Create another check
Test Check
Add Check

DATASET: prod_products — My Snowflake/prod_products
CHECK NAME: Valid publish date
SCHEDULE: Every day at midnight
COLUMN: country
CONDITION: contains
VALUE(S) - COMMA SEPARATED: US, BE
COLUMN: publish_date
VALIDITY RULE: valid format
VALUE: date us
ALERT LEVEL: Fail when
FAIL CONDITION: >
VALUE: 0
absolute

Figure 2.5: Soda Core GUI for check creation

Automated monitoring checks

The paid version of Soda, Soda Cloud, allows to add automated monitoring checks to a data source to monitor changes in it and tracking its evolution. Directly in the Soda Cloud console it can specified a monitoring check configuration like:

Listing 2.6: Example of Soda Cloud Monitoring Checks

```

automated monitoring:
  datasets:
    - include prod%
    - exclude test%
```

in the above example, it is defined an automated monitoring check which executes controls on all the datasets of which names begin with prod and excluding all the dataset with the name beginning with test

Chapter 3

Alertable

3.1 Introduction

Alertable is a Data Quality Accelerator framework, developed with the purpose of simplifying and centralizing the management of data quality check and alerts with an user friendly GUI. It has capabilities to monitor the freshness, the format, the distribution, the categories, the presence of outliers, and to notify if some alerts are triggered by the final user. The entire computation is developed with Apache Spark with AWS EMR for the cluster management, granting velocity and scalability and allowing us to separate the entire pipeline in a cloud AWS environment, accessible via API or other business tools.

3.2 Technologies

In this section are presented the principal technologies adopted by Alertable

3.2.1 Vaadin

Vaadin is an open-source web application development platform designed for creating rich, modern and responsive user interfaces. It offers a Java-based framework that enables developers to build single-page applications without needing deep knowledge of JavaScript, HTML, or CSS.
Vaadin Flow, its core framework, allows for server-side application logic. It also includes a comprehensive set of pre-built UI components, data binding tools, and seamless integration with Java backend technologies.[8]

3.2.2 Spring Boot

Spring Boot is a Java-based framework designed to simplify the development of standalone, production-ready applications. It builds on the Spring Framework by offering a preconfigured environment that reduces boilerplate code and accelerates development. Key features include auto-configuration, an embedded web server, and a convention-over-configuration approach.
Spring Boot supports a wide range of modules for database integration, security, messaging, and cloud deployment, making it ideal for microservices and modern distributed systems. Its main advantages are rapid setup, seamless integration with the Spring ecosystem, and flexibility for scaling applications.[9]

3.2.3 AWS EMR

AWS EMR (Elastic MapReduce) is a cloud-native big data platform provided by Amazon Web Services (AWS) that simplifies the processing and analysis of large datasets. It is built on top of popular open-source frameworks such as Apache Hadoop, Apache Spark, and Presto, making it highly scalable and suitable for diverse big data use cases, including data transformation, machine learning, log analysis, and real-time stream processing. The tool integrates completely within the AWS ecosystem and provide high efficiency, scalability and high availability with solid fault tolerance capability. [10]

3.3 Architecture

The architecture is composed by 5 components:

- **Pod Openshift:** it contains the docker image of the Java components for back end and front end
- **RDS Aurora:** it contains the metadata necessary for the working of Alertable and the results of the rules adopted
- **Secret Openshift:** where the references of Aurora Database are defined
- **Bucket S3:** it contains the details of the results and the logs produced by Alertable
- **Cluster EMR:** necessary for the computation of the Data quality rules

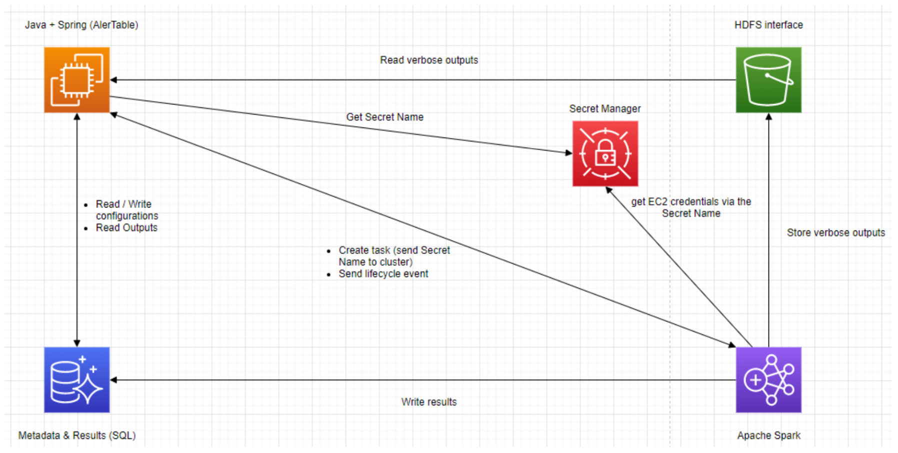


Figure 3.1: Alertable Architecture

3.4 Complete flow of Data Quality

1. **Datasource Import**: During this phase, it is possible to specify the connection parameters and import from the datasource some tables in order to make check on them.

2. **Check creation**: In this moment, it is defined the column level control on a specific table and it can be specified the test type.

3. **Alert creation**: Here it is defined an alert connected to a specific check and some conditions in order to define the constraints under which the check can be considered passed or not.

4. **Job Creation**: We can find here a list of different alerts to be executed and how the execution will be performed.

5. **Job Run**: After this phase, the job can be launched, and after its ending it is possible to see the results by visualizing some statistics referring to the alerts (if they were triggered) and some others referring to the checks (how many rows are failed, some relationships between the failed and the successful rows).

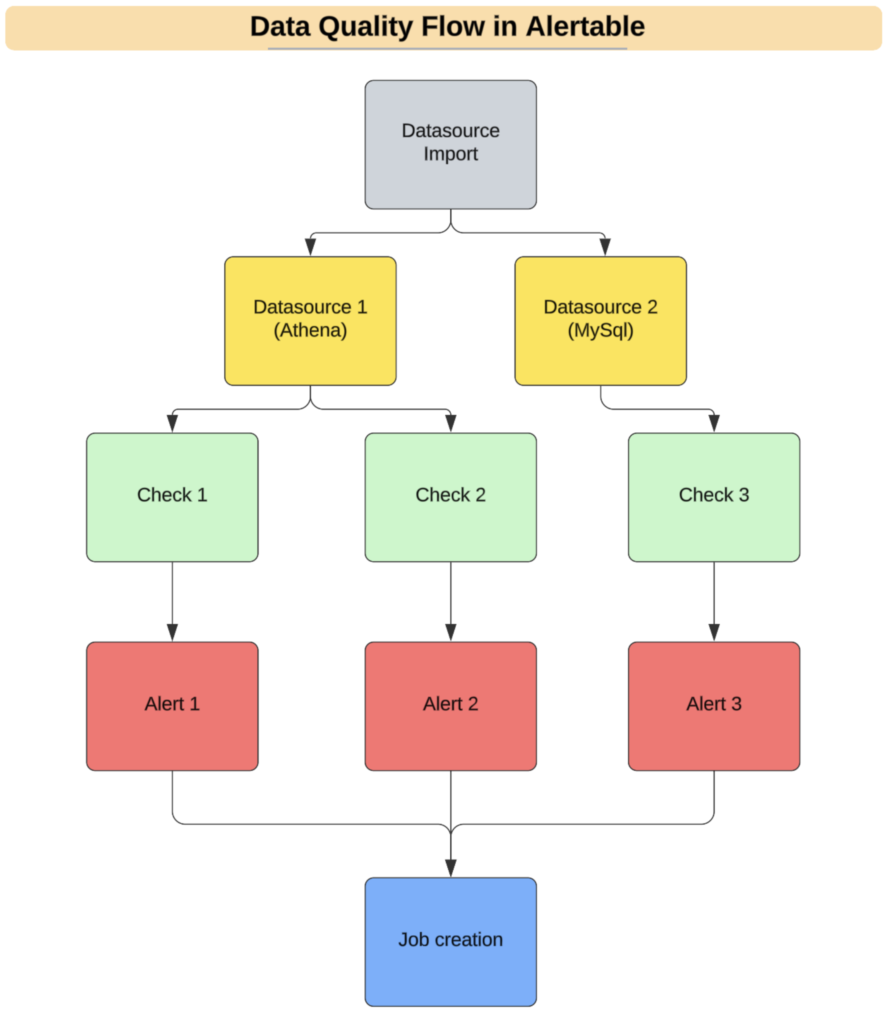


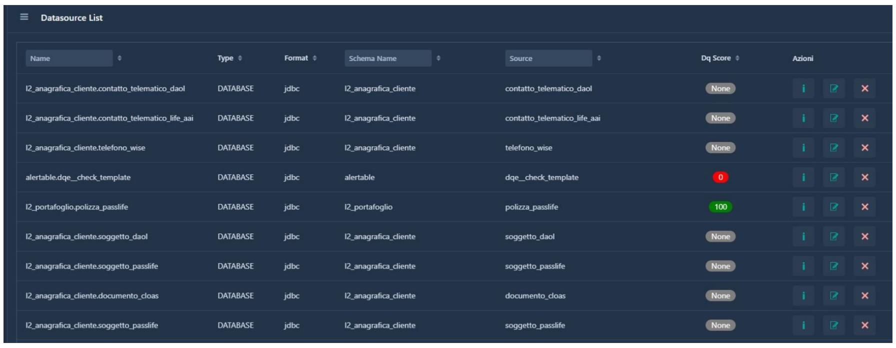
Figure 3.2: Data Quality Flow

3.5 Components

In this section are introduced the components featured in Alertable and shown their purpose, how they are defined and how they are integrated in the overall flow of the application.

3.5.1 Datasource

The first component is the Datasource, typically represented by a table, a view or a query on AWS Athena.



Datasource List

Name	Type	Format	Schema Name	Source	Dq Score	Azioni
l2_anagrafica_cliente.contatto_telematico_daol	DATABASE	jdbc	l2_anagrafica_cliente	contatto_telematico_daol	None	
l2_anagrafica_cliente.contatto_telematico_life_aai	DATABASE	jdbc	l2_anagrafica_cliente	contatto_telematico_life_aai	None	
l2_anagrafica_cliente.telefono_wise	DATABASE	jdbc	l2_anagrafica_cliente	telefono_wise	None	
alertable.dqe__check_template	DATABASE	jdbc	alertable	dqe__check_template	0	
l2_portafoglio.polizza_passlife	DATABASE	jdbc	l2_portafoglio	polizza_passlife	100	
l2_anagrafica_cliente.soggetto_daol	DATABASE	jdbc	l2_anagrafica_cliente	soggetto_daol	None	
l2_anagrafica_cliente.soggetto_passlife	DATABASE	jdbc	l2_anagrafica_cliente	soggetto_passlife	None	
l2_anagrafica_cliente.documento_cloas	DATABASE	jdbc	l2_anagrafica_cliente	documento_cloas	None	
l2_anagrafica_cliente.soggetto_passlife	DATABASE	jdbc	l2_anagrafica_cliente	soggetto_passlife	None	

Figure 3.3: Datasource list

Import of a datasource

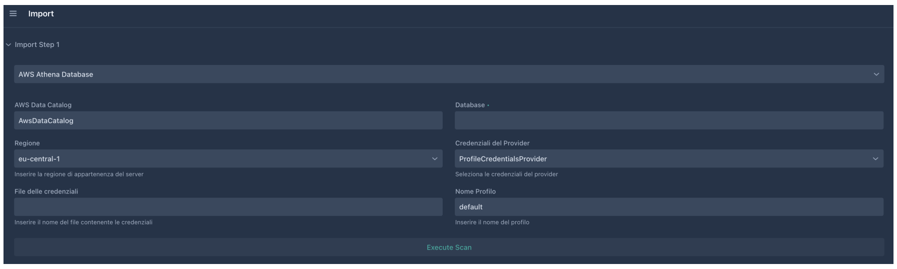
In the import of a datasource, there are multiple steps.
During the first one it is selected the kind of data source:

- **AWS Athena:** for which it has to be specified the AWS region, the name of the schema, the credentials provider, the profile name and the data catalog

- **AWS Athena Query:** which represents a view defined by a query on a Athena Database

- **Postgres SQL:** for which the connection is handled by JDBC

- **MySQL:** for which the connection is handled by JDBC



Import

Import Step 1

AWS Athena Database

AWS Data Catalog: AwsDataCatalog

Database •:

Regione: eu-central-1
Inserire la regione di appartenenza del server

Credenziali del Provider: ProfileCredentialsProvider
Seleziona le credenziali del provider

File delle credenziali:
Inserire il nome del file contenente le credenziali

Nome Profilo: default
Inserire il nome del profilo

Figure 3.4: Datasource Import Step 1

In the second step, after performing a scan of the database in order to retrieve all the tables in the datasource specified, it is shown a list of all the tables available and the possibility to select a list of them in order to import them in the app.

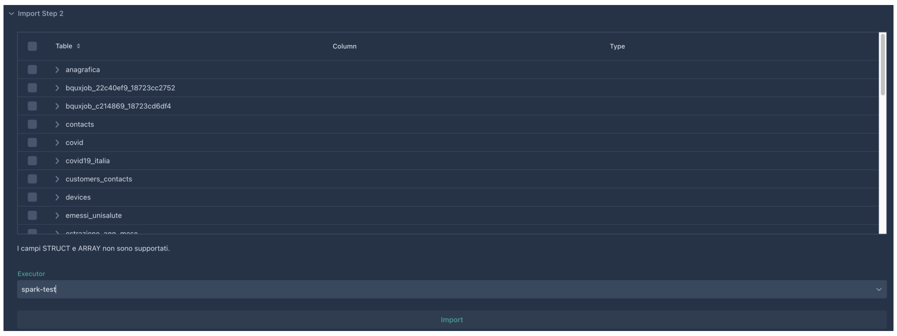

Import Step 2

Table	Column	Type
anagrafica		
bquxjob_22c40ef9_18723cc2752		
bquxjob_c214869_18723cd6df4		
contacts		
covid		
covid19_italia		
customers_contacts		
devices		
emessi_unisalute		

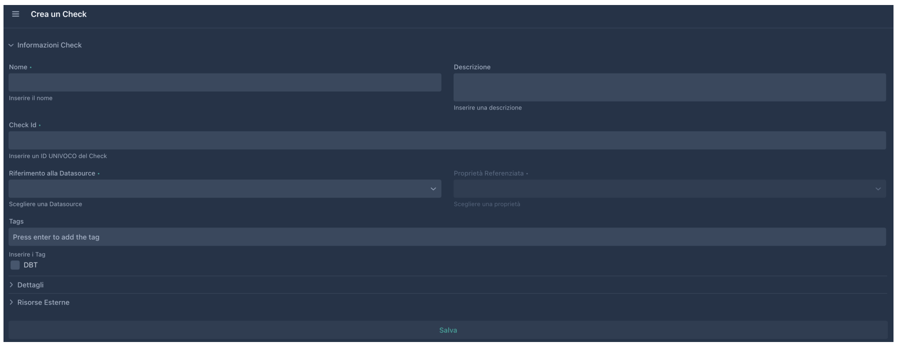
I campi STRUCT e ARRAY non sono supportati.

Executor: spark-test

Figure 3.5: Datasource Import Step 2

3.5.2 Checks

The Check represents the Data Quality Rule, which is composed by a filter and/or an aggregation. It can be defined for a specific property or column of a datasource, it possesses a skip condition in a SQL format for the rows that had to be skipped and the check can also be characterized by an external datasource for performing a join between tables.


Crea un Check

Informazioni Check

Nome •
Inserire il nome

Descrizione
Inserire una descrizione

Check Id •
Inserire un ID UNIVOCO del Check

Riferimento alla Datasource •
Scegliere una Datasource

Proprietà Referenziata •
Scegliere una proprietà

Tags
Press enter to add the tag
Inserire i Tag

DBT

Figure 3.6: Check creation

Filter

The filter defines a rule to be respected by a property, and it produces an output composed of two parts:

- A boolean flag for a single row to indicate if the rows has passed the control or not.
- A summary of the result, in which are defined the number of success or failure and some metrics.

Aggregation

The aggregation represents a calculation on an aggregated portion of the data, that can also be filtered if a filter is defined.

Check Template

The check template is a json object that represents the check parameters and their type.

Listing 3.1: Example of template for the check VALUES BETWEEN BOUNDARIES

```
{
  "type": "object",
  "required": ["lower_bound", "upper_bound"],
  "properties": {
    "lower_bound": { "type": ["number"] },
    "upper_bound": { "type": ["number"] }
  },
  "additionalProperties": false
}
```

3.5.3 Alerts

The alert has the objective to define a condition that triggers a notification and its kind.

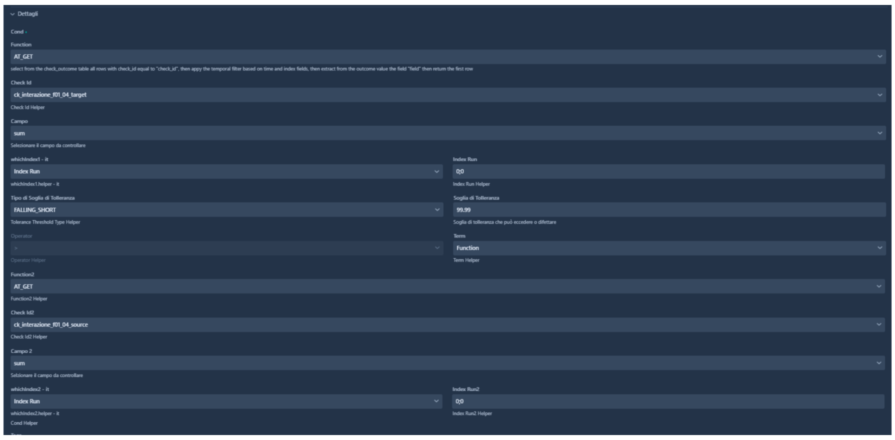

Dettagli
Cond
Function
AT_GET
select from the check_outcome table all rows with check_id equal to "check_id", then appy the temporal filter based on time and index fields, then extract from the outcome value the field "field" then return the first row
Check Id
ck_interazione_f01_04_target
Check Id Helper
Campo
sum
Selezionare il campo da controllare
whichIndex1 - it
Index Run
whichIndex1.helper - it
Index Run
0;0
Index Run Helper
Tipo di Soglia di Tolleranza
FALLING_SHORT
Tolerance Threshold Type Helper
Soglia di Tolleranza
99.99
Soglia di tolleranza che può eccedere o difettare
Operator
>
Operator Helper
Term
Function
Term Helper
Function2
AT_GET
Function2 Helper
Check Id2
ck_interazione_f01_04_source
Check Id2 Helper
Campo 2
sum
Selzionare il campo da controllare
whichIndex2 - it
Index Run
whichIndex2.helper - it
Index Run2
0;0
Index Run2 Helper
Cond Helper

Figure 3.7: Alert Creation

Alert Condition

Every alert has a check associated and it is characterized by a condition based on the output of the check where it can be specified:

- The field of the check output, like the count of success or failure.
- A retrieve function to extract the field (like a simple get or extract the maximum or minimum field).
- A tolerance threshold type to define how many records have to fail the condition in order to trigger the alert. It can be just a single number or another condition.

An example of condition can be the following, where it is defined an alert triggered if a column has duplicate values:

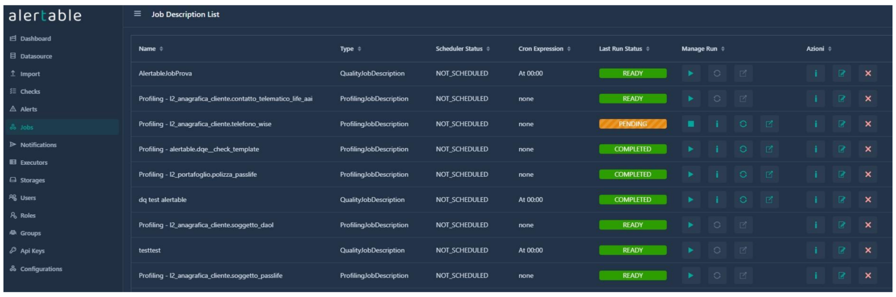
Listing 3.2: The condition retrieve the count of the unique values and check if it is minor or equal to the number of total rows.

```
AT_GET(
    check_ids="ck_unique_warehouseAndRetailSales_itemCode",
    field=count_unique
) <
AT_GET(
    check_ids="ck_unique_warehouseAndRetailSales_itemCode",
    field=count
)
```

3.5.4 Jobs

A job is an aggregation of different alerts, even belonging to different datasources, and it is used to define a procedure to aggregate different checks in order to manage the data quality rules.
It is possible to define some job parameters in order to dynamically personalize the job, the batch dimensions of the check performed contemporaneously, and a time schedule in order to define when and how frequently execute the job.
Furthermore, a job is also characterized by an executor which can be:

- **EMR Spark**: EMR (Elastic Map Reduce) is a managed service by AWS designed to simplify big data processing using distributed frameworks, like Apache Hadoop or Spark.
- **Local Spark**: It simply represents a Spark execution in local mode, so the process runs on a single machine instead of a cluster. It is typically used for local development and testing.

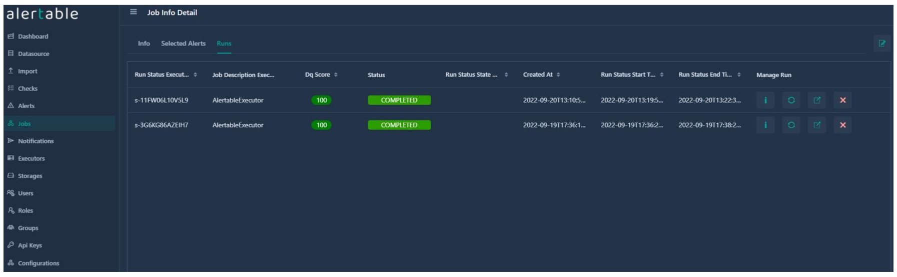


Name	Type	Scheduler Status	Cron Expression	Last Run Status	Manage Run	Azioni
AlertableJobProva	QualityJobDescription	NOT_SCHEDULED	At 00:00	READY		
Profiling - l2_anagrafica_cliente.contatto_telematico_life_aai	ProfilingJobDescription	NOT_SCHEDULED	none	READY		
Profiling - l2_anagrafica_cliente.telefono_wise	ProfilingJobDescription	NOT_SCHEDULED	none	PENDING		
Profiling - alertable.dqe__check_template	ProfilingJobDescription	NOT_SCHEDULED	none	COMPLETED		
Profiling - l2_portafoglio.polizza_passlife	ProfilingJobDescription	NOT_SCHEDULED	none	COMPLETED		
dq test alertable	QualityJobDescription	NOT_SCHEDULED	At 00:00	COMPLETED		
Profiling - l2_anagrafica_cliente.soggetto_daol	ProfilingJobDescription	NOT_SCHEDULED	none	READY		
testtest	QualityJobDescription	NOT_SCHEDULED	At 00:00	READY		
Profiling - l2_anagrafica_cliente.soggetto_passlife	ProfilingJobDescription	NOT_SCHEDULED	none	READY		

Figure 3.8: Job List

Job Result

Once the job is finished, it is possible to consult the results, see how many alerts have failed and also to update a DQ score - which is a number representing the overall quality of a datasource, based on the result of all the alerts applied to it.

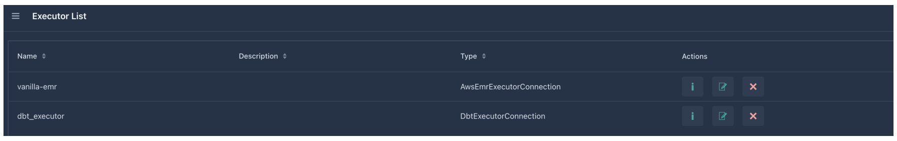


Run Status Execut...	Job Description Exec...	Dq Score	Status	Run Status State ...	Created At	Run Status Start T...	Run Status End Ti...	Manage Run
s-11FW06L10V5L9	AlertableExecutor	100	COMPLETED		2022-09-20T13:10:5...	2022-09-20T13:19:5...	2022-09-20T13:22:3...	
s-3G6KG86AZEIH7	AlertableExecutor	100	COMPLETED		2022-09-19T17:36:1...	2022-09-19T17:36:2...	2022-09-19T17:38:2...	

Figure 3.9: Job Result Summary

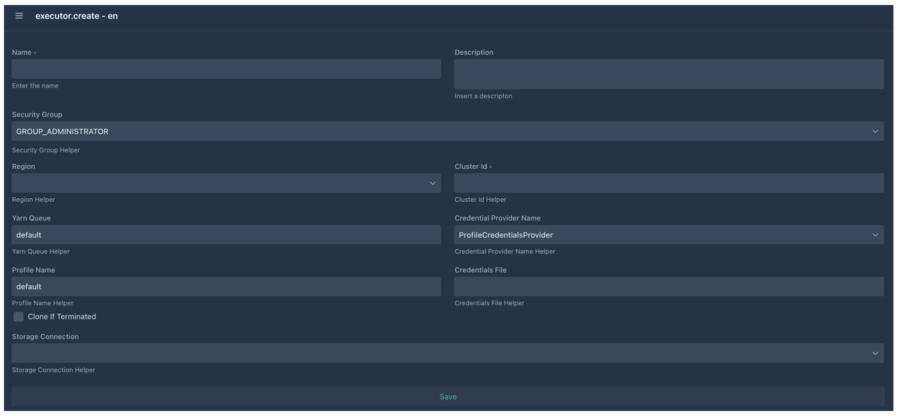
3.5.5 Executor

An executor specifies the execution type of a job.
It can be of various providers (like Google DataPrococ, Databricks or AWS EMR), and it allows to define some parameters depending on the type chosen, like the yarn queue, the cluster id or the credentials parameters in case of AWS.



Name	Description	Type	Actions
vanilla-emr		AwsEmrExecutorConnection	
dbt_executor		DbtExecutorConnection	

Figure 3.10: Executioner List



executor.create - en

Name •

Enter the name

Description

Insert a descripton

Security Group

GROUP_ADMINISTRATOR

Security Group Helper

Region

Region Helper

Cluster Id •

Cluster Id Helper

Yarn Queue

default

Yarn Queue Helper

Credential Provider Name

ProfileCredentialsProvider

Credential Provider Name Helper

Profile Name

default

Profile Name Helper

Credentials File

Credentials File Helper

☐ Clone If Terminated

Storage Connection

Storage Connection Helper

Figure 3.11: Executioner Creation

Chapter 4

DBT

4.1 Introduction to DBT

DBT (Data Build Tool) is a command-line tool that allows analytics engineers to transform data in data warehouses using SQL and Python. It introduces software engineering best practices to data transformation workflows.
It operates as a transformation layer in the data stack, sitting between the data warehouse and BI tools and it allows to write SQL models that transform raw data into analytics-ready datasets. Innovation is treating these transformations as modular, testable, and documentable code units, manageable from a single project.

DBT adopts Jinja as an SQL engine which increases the expressivity and the capabilities of SQL vanilla, allowing one to define reusable model and to write by using a more standard programmatic approach to the queries.
Another important aspect is the possibility of working on data models by versioning them, defining tests, and documenting them automatically; by doing so, it facilitates the deployment process into production by providing monitoring and visibility capabilities. DBT offers two different options in order to use it:

- **Dbt Core:** An open source tool that allows a manual set up for DBT project with a local management. This requires more work in order to organize and maintain a full DBT project but it offers full control on the structure and the deployment of DBT application.

- **DBT Cloud:** It is a non-free application that significantly extends the capabilities of DBT core by adopting an all-in-one-broswer-based UI, which provides a cloud CLI with auto suggestions and personalized commands to develop, test, run and version control DBT projects, a personalized IDE in order to define, build and test SQL models, enhanced with DBT Copilot -

which is an AI engine that generates code, documentation and checks the semantic of the models. In addition to that, DBT Cloud offers the possibility of schedule and run jobs, CI/CD pipelines, monitoring and alerting capabilities and also defining and managing multiple environments.

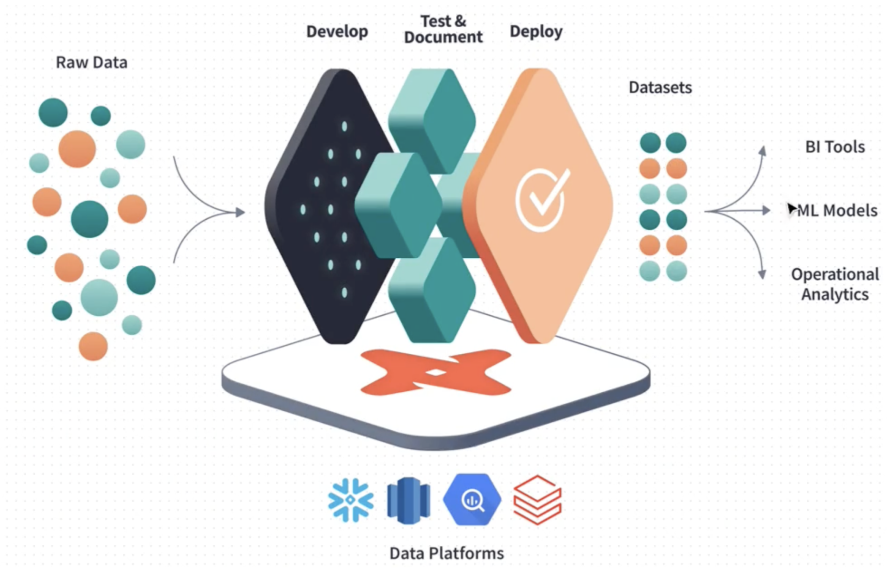


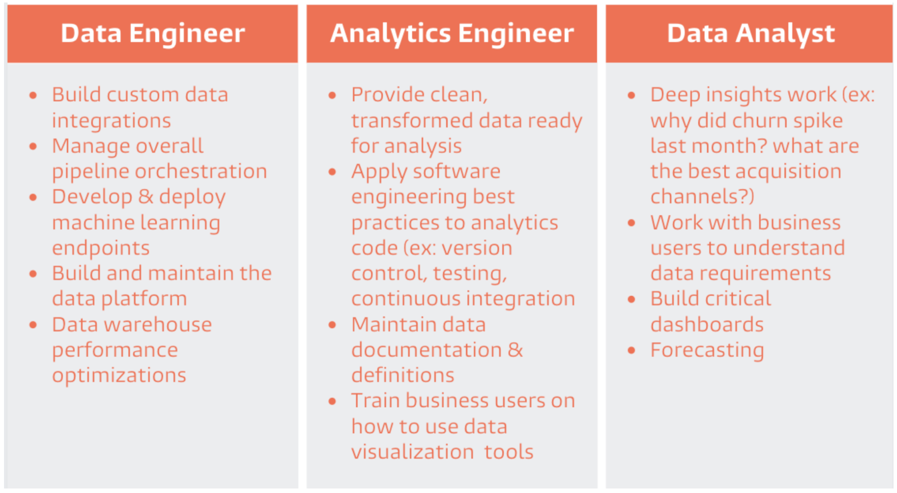
Figure 4.1: Dbt Role in ETL Architecture

4.2 The Role of DBT in Analytics Engineering

The analytical engineering is a new role [11] that has emerged in the recent years with the purpose of exploring the data already ingested in a platform to answer the questions of stakeholders, prepare and transform the data, applying a technique for cleaning and manipulating the data that can serve organizational objectives and also document all the objects created that can be found in a data warehouse to ensure clarity and usability.
It is a role emerged in order to help data engineering professionals in responding and transforming data for stakeholders, performing as bridge between tech and business roles. DBT is perfectly suited for that role, it only requires the knowledge of SQL to define and to write queries by offering a unified environment to work.

4.3 Use Cases and Practical Applications

DBT is a versatile tool suited for different scenarios, from data engineering to data analyst problems. It can help both in organizing, defining and tracking models and



Data Engineer	Analytics Engineer	Data Analyst
• Build custom data integrations • Manage overall pipeline orchestration • Develop & deploy machine learning endpoints • Build and maintain the data platform • Data warehouse performance optimizations	• Provide clean, transformed data ready for analysis • Apply software engineering best practices to analytics code (ex: version control, testing, continuous integration • Maintain data documentation & definitions • Train business users on how to use data visualization tools	• Deep insights work (ex: why did churn spike last month? what are the best acquisition channels?) • Work with business users to understand data requirements • Build critical dashboards • Forecasting

Figure 4.2: Analytical Engineering Overview [12]

in applying data quality check.
It also has capabilities in ensuring correct data pipelines and, eventually, showing the errors in it. Some common use cases are [13]:

4.3.1 Data Quality and Testing

DBT allows to clean data with simple SQL statements, allowing to build complex models to filter, transform and automate data tasks.
In order to avoid errors due to source changes, DBT allows to create sources so that it is possible to use them through a reference and whenever it is necessary to change the data source parameters, it has to change only the source definition while the code changes in the downstream processes can remain the same.

4.3.2 Documentation

DBT offers a special command called docs which allows to generate a project's documentation website with all the models, the data quality tests and the project model dependencies.
It is possible to specify a subgroup of models to produce the documentation needed and also to host automatically a sample website, allowing the specification of a port.

4.3.3 Data lineage

In DBT the data lineage is handled with a Directed Acyclic Graph for tracking all the dependencies and links between models, transformations and tests.
It is helpful also in the definition and the maintenance of data pipelines with the help of singular and generic tests, that are Yaml or SQL files in the test directory of a DBT project.

4.4 DBT Commands

This section will show the principal commands adopted in a DBT project along with their functionality. [14]

4.4.1 DBT init

The init commands is required in order to initialize and create a dbt-core project. It requires the following parameters:

- The **project name**
- The **database adapter**
- **Connection parameters** depending on the adapter chosen

After the information request, a new folder will be created with the project name and all the principal components of a typical DBT project.
In addition to that it is created a connection profile on the local machine, usually in the location /.dbt/profiles.yml. It is possible to specify before launching the command a flag (–profile profileName) to indicate an existing profiles.yml file instead of creating a new one.

4.4.2 DBT test

The test commands run data tests defined on models, sources or unit tests but expects that these resources have already been created. It is possible to simply launch the command without a flag to execute all the present tests in the projects, belonging to the predefined target, or it is also possible to specify some flags, in order to restrict the tests launched by using various commands:

Listing 4.1: Example of dbt test commands

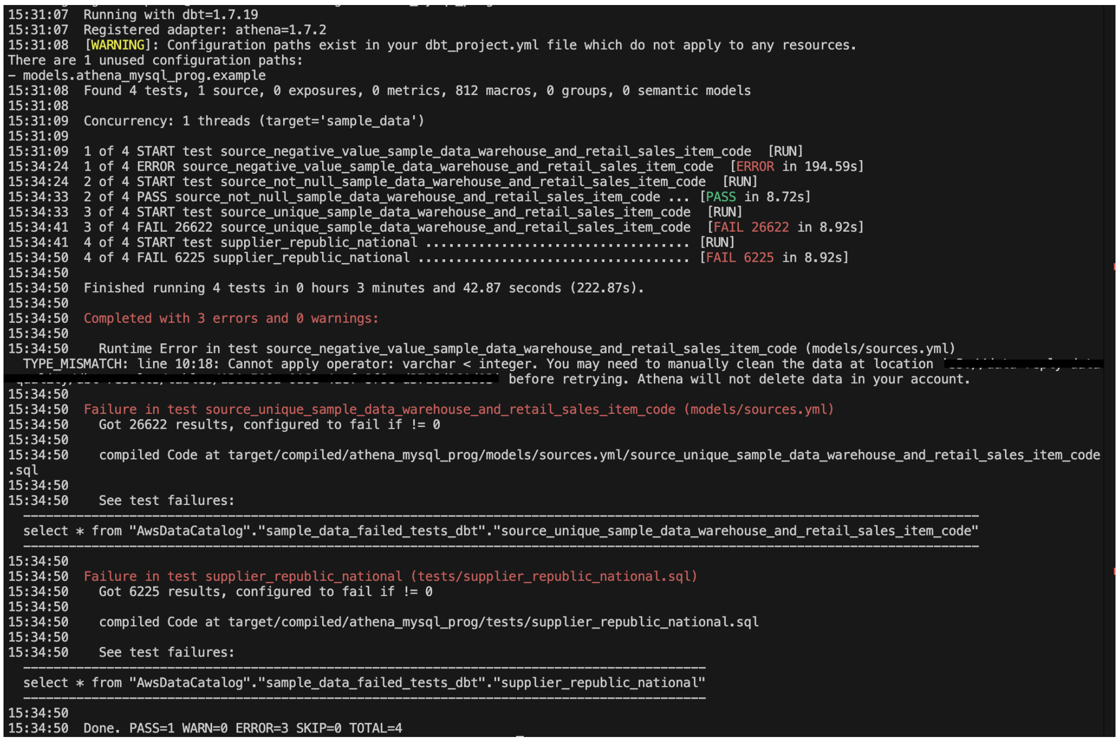
```
# run data and unit tests
dbt test

# run only data tests
dbt test --select test_type:data

# run only unit tests
dbt test --select test_type:unit

#run tests for all models with the tag specified
dbt test --select tag:tag_1

# run tests for one_specific_model
dbt test --select "one_specific_model"

# run tests for all models in package
dbt test --select "some_package.*"

# run only data tests defined singularly
dbt test --select "test_type:singular"

# run only data tests defined generically
dbt test --select "test_type:generic"

# run data tests limited to one_specific_model
dbt test --select "one_specific_model,test_type:data"

# run unit tests limited to one_specific_model
dbt test --select "one_specific_model,test_type:unit"
```

In addition to this flag it is possible to use the following one:

Listing 4.2: Example of dbt test specific flags

```

# run data test specifyng the target
dbt test --target sample_data

# run data test specifyng the number of threads
dbt test --threads 4
```

The possible results of a test can be PASS, FAIL or ERROR, as shown in the below image that also shows the typical output of the command. 4.3



```
15:31:07  Running with dbt=1.7.19
15:31:07  Registered adapter: athena=1.7.2
15:31:08  [WARNING]: Configuration paths exist in your dbt_project.yml file which do not apply to any resources.
There are 1 unused configuration paths:
- models.athena_mysql_prog.example
15:31:08  Found 4 tests, 1 source, 0 exposures, 0 metrics, 812 macros, 0 groups, 0 semantic models
15:31:08
15:31:09  Concurrency: 1 threads (target='sample_data')
15:31:09
15:31:09  1 of 4 START test source_negative_value_sample_data_warehouse_and_retail_sales_item_code  [RUN]
15:34:24  1 of 4 ERROR source_negative_value_sample_data_warehouse_and_retail_sales_item_code  [ERROR in 194.59s]
15:34:24  2 of 4 START test source_not_null_sample_data_warehouse_and_retail_sales_item_code  [RUN]
15:34:33  2 of 4 PASS source_not_null_sample_data_warehouse_and_retail_sales_item_code ... [PASS in 8.72s]
15:34:33  3 of 4 START test source_unique_sample_data_warehouse_and_retail_sales_item_code  [RUN]
15:34:41  3 of 4 FAIL 26622 source_unique_sample_data_warehouse_and_retail_sales_item_code  [FAIL 26622 in 8.92s]
15:34:41  4 of 4 START test supplier_republic_national ................................... [RUN]
15:34:50  4 of 4 FAIL 6225 supplier_republic_national .................................... [FAIL 6225 in 8.92s]
15:34:50
15:34:50  Finished running 4 tests in 0 hours 3 minutes and 42.87 seconds (222.87s).
15:34:50
15:34:50  Completed with 3 errors and 0 warnings:
15:34:50
15:34:50    Runtime Error in test source_negative_value_sample_data_warehouse_and_retail_sales_item_code (models/sources.yml)
  TYPE_MISMATCH: line 10:18: Cannot apply operator: varchar < integer. You may need to manually clean the data at location [redacted] before retrying. Athena will not delete data in your account.
15:34:50
15:34:50  Failure in test source_unique_sample_data_warehouse_and_retail_sales_item_code (models/sources.yml)
15:34:50    Got 26622 results, configured to fail if != 0
15:34:50
15:34:50    compiled Code at target/compiled/athena_mysql_prog/models/sources.yml/source_unique_sample_data_warehouse_and_retail_sales_item_code
.sql
15:34:50
15:34:50    See test failures:
  -----------------------------------------------------------------------------------------------------------------------------
  select * from "AwsDataCatalog"."sample_data_failed_tests_dbt"."source_unique_sample_data_warehouse_and_retail_sales_item_code"
  -----------------------------------------------------------------------------------------------------------------------------
15:34:50
15:34:50  Failure in test supplier_republic_national (tests/supplier_republic_national.sql)
15:34:50    Got 6225 results, configured to fail if != 0
15:34:50
15:34:50    compiled Code at target/compiled/athena_mysql_prog/tests/supplier_republic_national.sql
15:34:50
15:34:50    See test failures:
  ------------------------------------------------------------------------------------------
  select * from "AwsDataCatalog"."sample_data_failed_tests_dbt"."supplier_republic_national"
  ------------------------------------------------------------------------------------------
15:34:50
15:34:50  Done. PASS=1 WARN=0 ERROR=3 SKIP=0 TOTAL=4
```

Figure 4.3: Dbt Test Output

4.4.3 DBT run

The DBT run commands is used to execute and create all the SQL models featured in the models directory by following the dependency order. It accepts some flags like –select to restrict the execution.

4.4.4 DBT build

The DBT build commands run models and apply the tests following the order of the dependencies in DAG order and it represents a commands to aggregate both test and run commands, fo this reason it is used to check the overall coherence and correctness of a DBT project.
Following the DAG order means that if a model B depends on a model A, but some tests about model A fail, model B will be skipped.
As usual, it is possible to specify some flags, like –select, in order to restrict the build to certain data resources.

4.4.5 DBT deps

The DBT deps pulls the most recent version of the dependencies listed in the packages.yml file from GIT. It is used in order to import or update the dependencies library used in a project.

4.5 Anatomy of a DBT Core Project

In this section, it will be analysed the structure of a DBT core project, explaining the role of each folder and files and how they are used, especially in the context of the work in chapter 5.

4.5.1 Directory Structure

When launching the command DBT init, the typical structure of folders created is like the one below:

dbt_project.yml

This file contains all the general configuration of a DBT project, like the project name, the paths of DBT models and tests. In addition to that, it contains the table name where it is possible to memorize in the database the results of the failed tests; this aspect will be used in chapter 5 in order to retrieve the test.

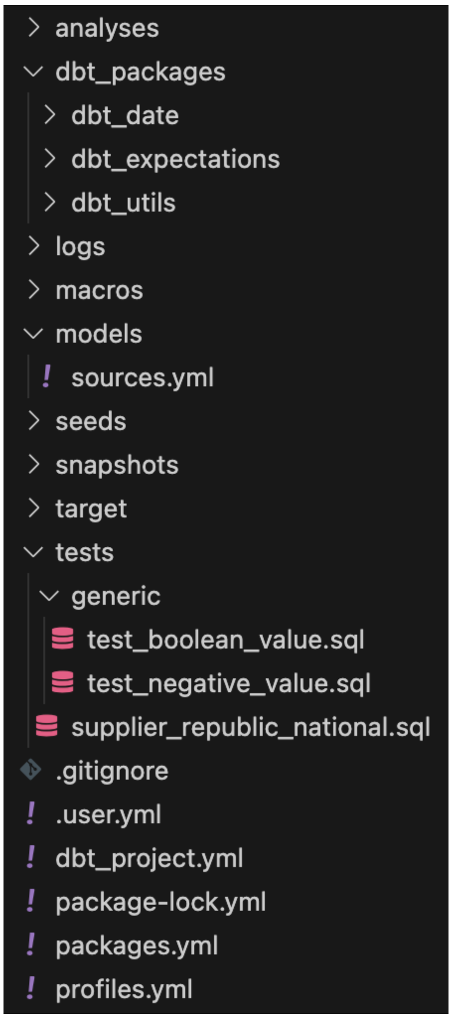
profiles.yml

This file contains the connection parameters to the datasources of the project. In each datasource component are specified different parameters depending on the connection type (Athena, MySql, ...) and also a predefined target datasource that is adopted during a DBT command, if there are no other specifications.
In the changes presented in Chapther 5, this file is used to save or update the datasource added in Alertable.

Models

In this folder are defined all the models adopted in a DBT project as SQL files where. There are 2 principal ways to define a model:

- **SQL Query:** the model is defined as a query where in the "from" clause is specified a source model or an intermediate one, defined somewhere else. Typically this definition is adopted to define the transformation layer and stages of the pipeline, where the data manipulation are applied directly in the query.



```
> analyses
v dbt_packages
  > dbt_date
  > dbt_expectations
  > dbt_utils
> logs
> macros
v models
  ! sources.yml
> seeds
> snapshots
> target
v tests
  v generic
    test_boolean_value.sql
    test_negative_value.sql
  supplier_republic_national.sql
.gitignore
! .user.yml
! dbt_project.yml
! package-lock.yml
! packages.yml
! profiles.yml
```

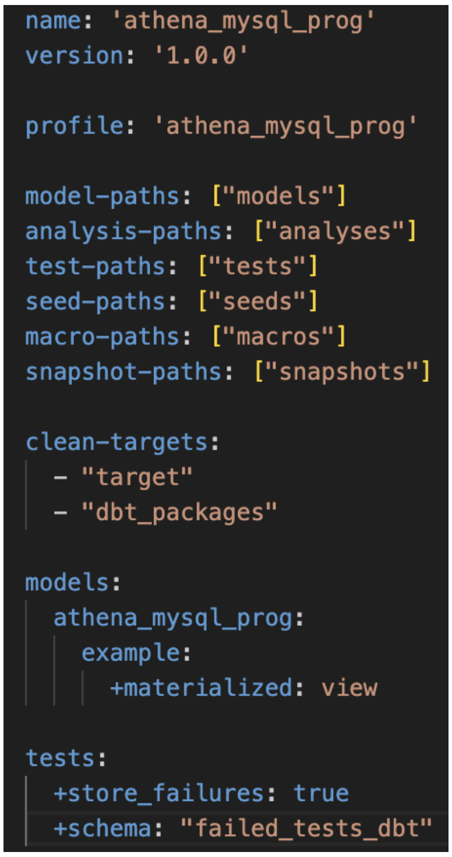
Figure 4.4: Dbt Folder Structure

- **YAML File:** the model is defined with its schema. With this format, the model can represent the initial stage, like the sources models or the final stage. Alongside the schema, in the YAML file are also defined its test, that can be at a columnar or tabular level. 4.7

Chapter 5 will show that, inside the folder, it is memorized the file sources.yml to track all the tables and the tests present in Alertable.

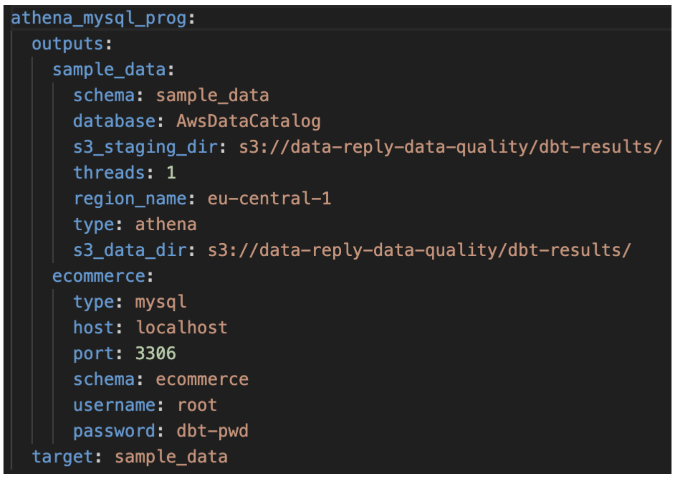
Tests

In the folder, tests are defined as test functions in addition to the built in DBT. In there, a test is defined as a SQL query and there are two kind of tests:



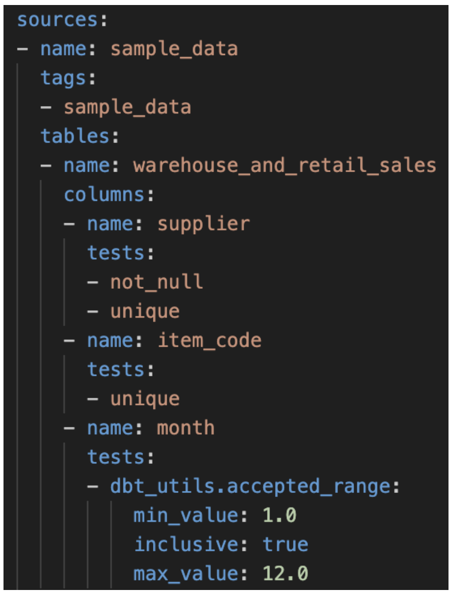
```yaml
name: 'athena_mysql_prog'
version: '1.0.0'

profile: 'athena_mysql_prog'

model-paths: ["models"]
analysis-paths: ["analyses"]
test-paths: ["tests"]
seed-paths: ["seeds"]
macro-paths: ["macros"]
snapshot-paths: ["snapshots"]

clean-targets:
  - "target"
  - "dbt_packages"

models:
  athena_mysql_prog:
    example:
      +materialized: view

tests:
  +store_failures: true
  +schema: "failed_tests_dbt"
```

Figure 4.5: Dbt Project File



```yaml
athena_mysql_prog:
  outputs:
    sample_data:
      schema: sample_data
      database: AwsDataCatalog
      s3_staging_dir: s3://data-reply-data-quality/dbt-results/
      threads: 1
      region_name: eu-central-1
      type: athena
      s3_data_dir: s3://data-reply-data-quality/dbt-results/
    ecommerce:
      type: mysql
      host: localhost
      port: 3306
      schema: ecommerce
      username: root
      password: dbt-pwd
  target: sample_data
```

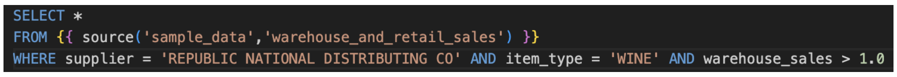
Figure 4.6: Dbt Profiles File



```yaml
sources:
- name: sample_data
  tags:
  - sample_data
  tables:
  - name: warehouse_and_retail_sales
    columns:
    - name: supplier
      tests:
      - not_null
      - unique
    - name: item_code
      tests:
      - unique
    - name: month
      tests:
      - dbt_utils.accepted_range:
          min_value: 1.0
          inclusive: true
          max_value: 12.0
```

Figure 4.7: Dbt Source Model File

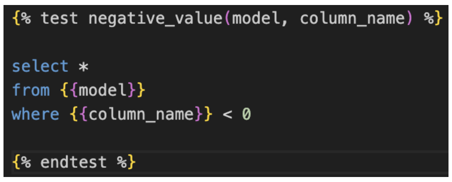
- **Custom test:** it is simply defined as a query where the “FROM" clause points to a fixed source and adopts fixed values in the where clause. 4.8



```sql
SELECT *
FROM {{ source('sample_data','warehouse_and_retail_sales') }}
WHERE supplier = 'REPUBLIC NATIONAL DISTRIBUTING CO' AND item_type = 'WINE' AND warehouse_sales > 1.0
```

Figure 4.8: Dbt Custom Test

- **Generic test:** is a SQL query function that can parametrically receive one or several sources on which it is possible to apply the test and also other parameters for test condition. 4.9



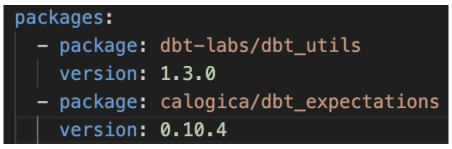
```sql
{% test negative_value(model, column_name) %}

select *
from {{model}}
where {{column_name}} < 0

{% endtest %}
```

Figure 4.9: Dbt Generic Tests

In Chapther 5 it will be shown that inside the folder there are also the generic test file, statically added as possible test, and the custom test dynamically added by a user in order to make its custom test.

packages.yml

In this file there are the packages added to the DBT project, in order to extend its functionalities. The packages, after they have been added, can be downloaded with the command DBT deps. 4.10



```
packages:
  - package: dbt-labs/dbt_utils
    version: 1.3.0
  - package: calogica/dbt_expectations
    version: 0.10.4
```

Figure 4.10: Dbt Packages

target

Inside this folder are added the executable SQL files that are generated by DBT during the commands DBT test, DBT run and DBT build and that will be used for the execution in the database

4.6 Comparison with Great Expectations and Soda Core

In this section DBT will be compared to Great Expectations and Soda Core through several aspects, from main goals to architecture and integration ease.

4.6.1 Comparison Criteria

To provide a thorough and meaningful comparison of DBT, Great Expectations, and Soda Core, it is essential to evaluate their core functionalities based on specific criteria. These include their purpose and focus, how seamlessly they integrate into modern data stacks, their scalability when handling large datasets and their flexibility in addressing complex use cases or adapting to specific requirements.

While DBT specializes in SQL-centric transformations and modeling, Great Expectations excels in validating and profiling data with precision, and Soda Core primarily aims to monitor data quality in real time, albeit with fewer generalized capabilities.

Integration is a critical factor in modern workflows, where tools must work seamlessly with cloud-native platforms and pipelines. Effective integration minimizes operational overhead and grants a smooth execution across different stages of data management.
Furthermore, scalability is a paramount for tools operating on large datasets, as

inefficient processing or limited parallelization can create bottlenecks. Lastly, flexibility encompasses a tool's ability to support advanced features, custom logic and extensibility, making it suitable for diverse scenarios.

Purpose and Focus

Each tool is suited to address a specific aspect of data management. DBT focuses on transforming and modeling data efficiently by leveraging SQL workflows, making it ideal for standardizing and structuring data before analysis.
Great Expectations, on the other hand, emphasizes detailed data validation and profiling, it is often used for complex data quality checks that require Python's flexibility and power.
Soda Core fills a niche by offering lightweight, real-time monitoring for detecting anomalies and inconsistencies, with a primary focus on simplicity and rapid deployment.

Ease of Use

The usability of these tools depends significantly on the expertise of the user. DBT and Soda Core stand out for their intuitive interfaces. DBT is designed for data analysts and engineers comfortable with SQL, allowing them to build modular and maintainable models.
Similarly, Soda Core uses a YAML-based configuration system that simplifies the setup for real-time monitoring, requiring minimal programming knowledge.

Conversely, Great Expectations, while more powerful for complex validation scenarios, demands a deeper understanding of Python. This makes it better suited for data engineers or teams with significant programming expertise.
Consequently, DBT and Soda Core are generally more accessible to SQL users, while Great Expectations caters to those needing advanced validation logic and greater control over their workflows.

Integration and Ecosystem

The ability to easily integrate into existing data ecosystems is a defining characteristic of modern tools. DBT integrates natively with major cloud data warehouses such as Snowflake, BigQuery and Redshift, requiring the inclusion of a simple adapter to connect to these platforms.
This integration streamlines workflows and ensures compatibility with cloud-native architectures.
Great Expectations provides broader support by working with multiple backends, including Pandas for small datasets, Spark for distributed processing, and SQL

databases for relational data. This versatility allows it to operate across various environments, from local development to large-scale distributed systems.

Soda Core, while less versatile in its integrations, is better for its simplicity and focus. It is designed to fit seamlessly into CI/CD pipelines, enabling anomaly detection during automated deployments with minimal configuration effort.

Scalability and Efficiency

Scalability is a crucial consideration when selecting tools for large datasets. DBT, relying on the computational power of the underlying data warehouse, scales exceptionally as well as it delegates heavy lifting to platforms like Snowflake or BigQuery.
This design ensures high performance even when processing terabytes of data, as the transformation logic leverages the warehouse's inherent parallelization capabilities.

Great Expectations, while powerful, depends heavily on the backend being used. For example, when paired with Spark, it can handle distributed datasets efficiently, but its performance may be limited when using Pandas for large data volumes. Its scalability is therefore closely tied to the underlying infrastructure.

Soda Core, focused on lightweight monitoring, is optimized for smaller datasets or real-time streams. While efficient for its intended use cases, it may not be the best choice for scenarios involving massive batch processing or complex data transformations.

4.6.2 Final Considerations

In order to be considered the integration in Alertable, the main aspects contributing are the scalability and the simplicity of integration. It has to be considered that Alertable already handles the job schedule and the data profiling , it is more focused on the data quality of the database and data warehouse, making Soda Core less useful.
Despite Great Expectations provides a broader suite of tests, DBT can incorporate them with the dbt_expectations and, also, it posses a simple and intuitive way to connect the most major data warehouse with the connectors.

Chapter 5

DBT Integration in Alertable

5.1 Technologies

In this section there will be listed all the principal technologies involved and adopted for the DBT integration in Alertable, from the cloud provider which is AWS, to the web app technologies, like Spring and Vaadin.

5.1.1 Aws EC2

Amazon EC2 (Elastic Compute Cloud) is a scalable, pay-as-you-go cloud computing service provided by AWS that offers virtual servers, known as instances.
It allows users to select from a wide range of instance types optimized for compute, memory, or storage needs, with flexibility in operating systems and configurations. The EC2 instances allows to scale up or down based on demand and can distribute incoming traffic across multiple instances to handle variable workloads.
The instance used for the Alertable application in this work is of type t2.medium with Linux as operating system. [15]

5.1.2 Aws ECR

Amazon Elastic Container Registry (ECR) is a fully managed container image registry by AWS - designed for storing, managing, and deploying Docker container images securely at scale.
It simplifies the deployment of containerized applications. It supports private and public repositories, with image versioning, automated lifecycle policies for image cleanup, and vulnerability scanning for enhanced security. During the work the

service was used in order to deploy the EC2 instance with the specified docker image.[16]

5.1.3 Aws Athena

Amazon Athena is an interactive serverless query service from AWS that allows users to analyze data directly in Amazon S3 using standard SQL.
It eliminates the need for infrastructure management and scales automatically to handle complex queries.

Athena supports structured, semi-structured and unstructured data in various formats, including CSV, JSON or Parquet.
With its ability to work with large datasets, Athena is an ideal choice for analysis, business intelligence tasks, and exploratory data processing, ensuring fast and efficient performances. [17] This datasource is the principal database on which Alertable works and it was used to provide the tables and the database environment on which it is possible to run the queries.

5.1.4 Aws RDS

Amazon RDS (Relational Database Service) is a managed database service from AWS that simplifies the process of setting up, operating and scaling relational databases in the cloud.
With it, users can offload administrative tasks such as hardware provisioning, software patching, backup management, and database scaling, this allows them to focus on application development. Additionally, RDS supports performance optimization with features like read replicas and customizable instance types, making it suitable for a wide range of applications, from small-scale web apps to large enterprise systems. [18]

5.1.5 Aws S3

Amazon S3 (Simple Storage Service) is a highly scalable, secure and durable object storage service provided by AWS. It allows you to store and retrieve any amount of data from anywhere on the Web.
S3 is designed for a variety of use cases, including data backup, content storage and distribution, big data analytics and disaster recovery. [19]

5.1.6 Docker

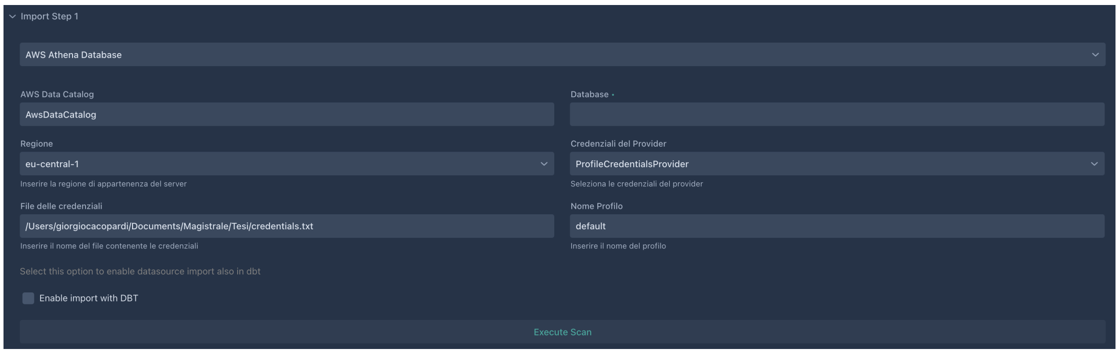
Docker is an open-source platform for developing, shipping and running applications in lightweight, isolated containers. It enables developers to package applications with all dependencies, ensuring consistent behavior across environments. It operates on the basis of images, which are templates for creating containers. These containers are portable, scalable, and efficient, using fewer resources than virtual machines.[20] Docker was used in order to update the Alertable image to include all the necessary dependencies and instructions to integrate DBT.

5.2 Import of datasource

The first step for DBT integration is to handle the import of the database in the application giving the user the possibility to decide whether to add the datasource or not.
In order to do so, it was added a checkbox option to select the preference. In some cases, like Athena datasource, it was necessary to dynamically add some options required by DBT for the connections. An option was also added to directly import some tests written in a YAML file.
Overall, this results in three different flows during the import phase:

1. **Flow without DBT**, with the checkbox for including dbt selected to false, as shown in 5.1

    a) Specifications of connection parameters, as shown in 3.4



Import Step 1

AWS Athena Database

AWS Data Catalog: AwsDataCatalog

Database •:

Regione: eu-central-1
Inserire la regione di appartenenza del server

Credenziali del Provider: ProfileCredentialsProvider
Seleziona le credenziali del provider

File delle credenziali: /Users/giorgiocacopardi/Documents/Magistrale/Tesi/credentials.txt
Inserire il nome del file contenente le credenziali

Nome Profilo: default
Inserire il nome del profilo

Select this option to enable datasource import also in dbt

☐ Enable import with DBT

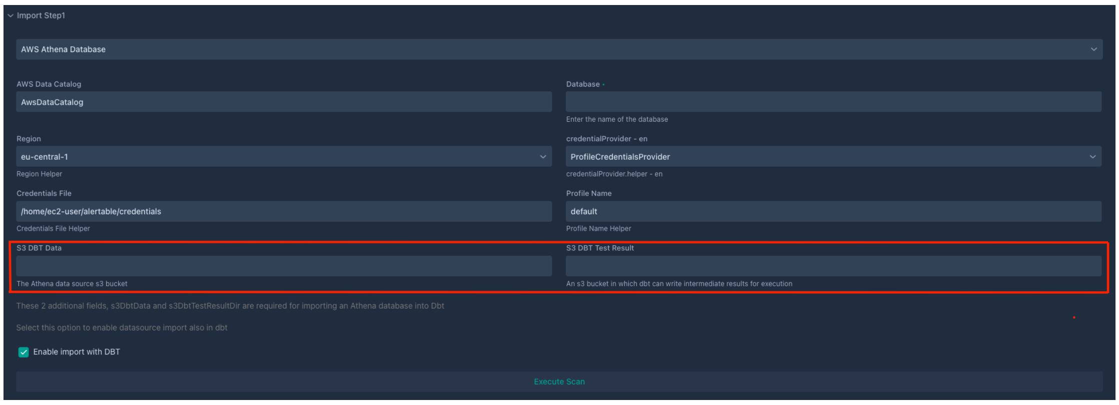
Figure 5.1: Import Datasource Step 1 With DBT

    b) Selection of tables for import, as shown in 3.5

2. **Flow with DBT** with the checkbox enabled. In this case,

    a) specifications of connection parameters but with optional fields in the form depending on the necessity of the DBT connector for the specific type of the database, as shown in 5.2

    b) dropdown list with 2 choice 5.3


Import Step1

AWS Athena Database

AWS Data Catalog: AwsDataCatalog

Database:
Enter the name of the database

Region: eu-central-1
Region Helper

credentialProvider - en: ProfileCredentialsProvider
credentialProvider.helper - en

Credentials File: /home/ec2-user/alertable/credentials
Credentials File Helper

Profile Name: default
Profile Name Helper

S3 DBT Data:
The Athena data source s3 bucket

S3 DBT Test Result:
An s3 bucket in which dbt can write intermediate results for execution

These 2 additional fields, s3DbtData and s3DbtTestResultDir are required for importing an Athena database into Dbt

Select this option to enable datasource import also in dbt

✓ Enable import with DBT

Figure 5.2: Import Datasource Step 1 With Dbt Flagged

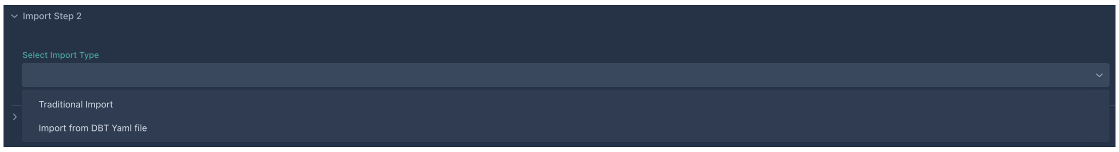
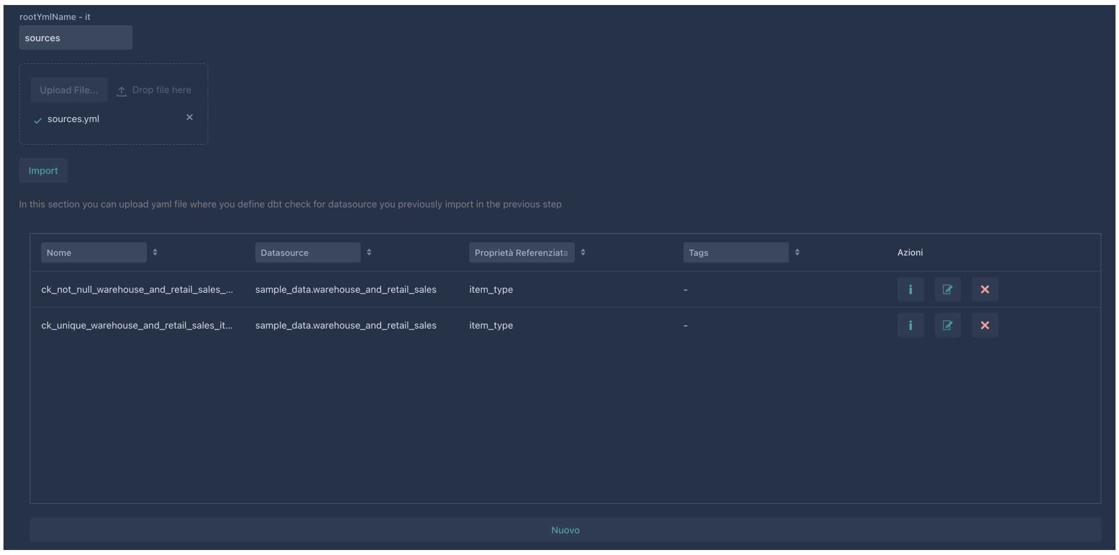


Figure 5.3: Import Datasource Step 2 Dropdown Menu

a) Selection of tables as shown in 3.5

b) Import of check and tables from a yaml file, similar to 4.7, as shown in 5.4


rootYmlName - it: sources

✓ sources.yml

In this section you can upload yaml file where you define dbt check for datasource you previously import in the previous step

Nome	Datasource	Proprietà Referenziata	Tags	Azioni
ck_not_null_warehouse_and_retail_sales_...	sample_data.warehouse_and_retail_sales	item_type	-	
ck_unique_warehouse_and_retail_sales_it...	sample_data.warehouse_and_retail_sales	item_type	-	

Figure 5.4: Import Datasource Step 2 With Import from file upload

5.3 Check creation

In the check creation component,3.5.2 it has to be changed the way a check is configured, adapting it to be integrable with DBT.
The possible tests are the following:

- **Built-in Tests**
  - **not null**
  - **unique**
  - **accepted values**: this test checks if a value belongs to one of the elements of an array passed as a parameter.
- **Package Tests**: The following tests are imported from the libraries *dbt-utils* and *dbt-expectations*.
  - **dbt-utils.at_least_one**: this test checks if at least one value in a specific column, considering all the rows, is not null.
  - **dbt-utils.accepted_range**: this test checks if a numerical value stays within an interval defined by 3 parameters (`max_value`, `min_value`, and `inclusive`, to define if the interval is open or closed).
  - **dbt_expectations.expect_column_value_lengths_to_equal**: this test checks if a string has the specified length.
- **Generic Tests (4.9)**:
  - **negative value**
  - **boolean value**
- **Custom Test (4.8)**
  - **custom Where**: this test is defined by the user and is a query with a customizable where clause.

A dynamic field form is shown in case the check type is DBT in order to match the different parameters required by each test (if present). Two example are shown for accepted_range and custom_where tests respectively in 5.6 and 5.7

5.4 DBT Job Executor Architecture

The architecture of a DBT Job is structured in three main steps:

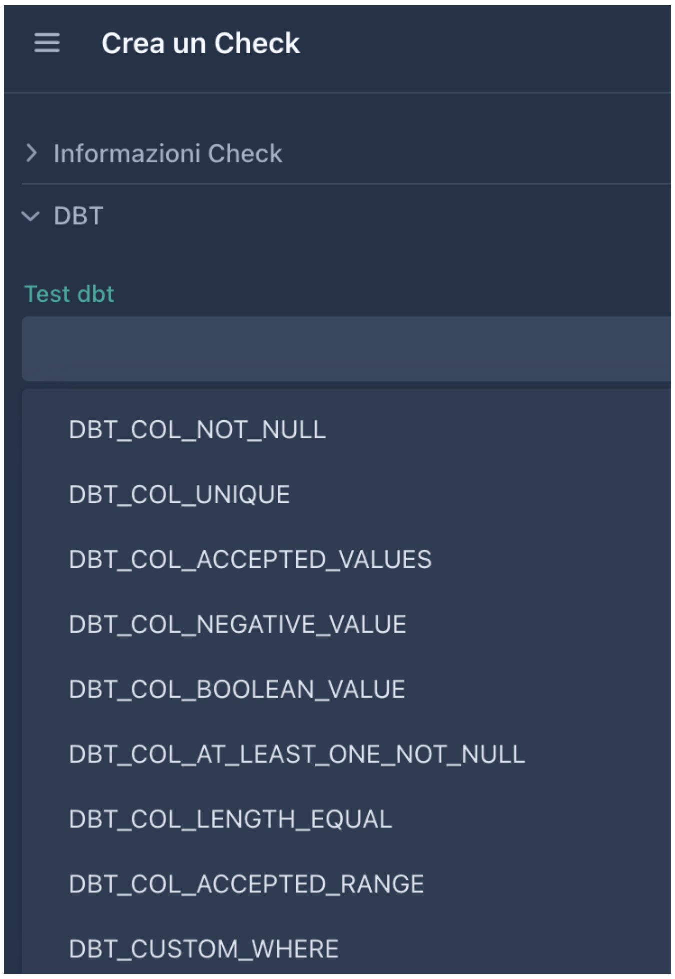


Figure 5.5: DBT Check Form Selection

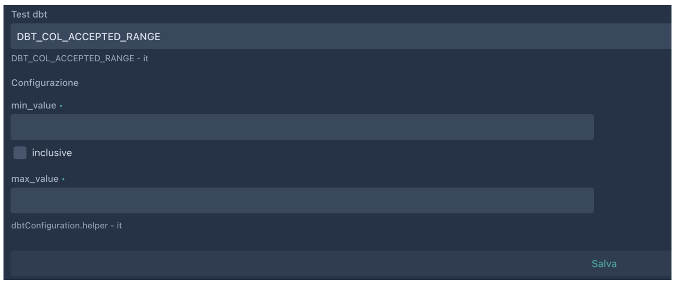


Figure 5.6: DBT Accepted Range Form

1. The writing of all the checks belonging to the job in the sources file of the DBT project

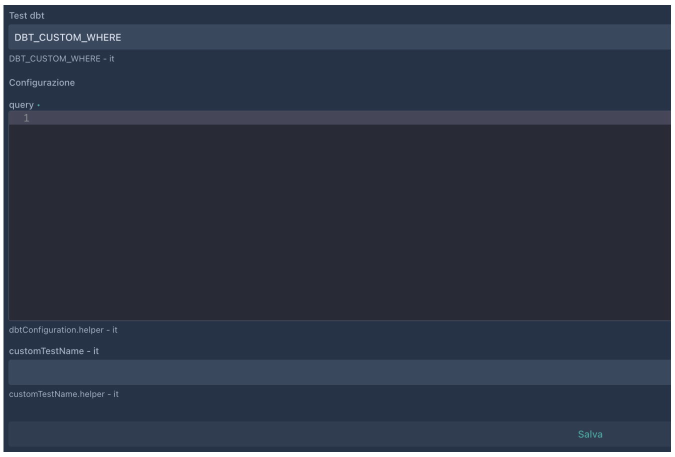


Figure 5.7: DBT Custom Where Form

2. The decomposition in multiple processes, each executing a DBT test command for a specific datasource
3. The retrieve of all the processes outputs with all the results for every checks

5.4.1 Writing in sources file in DBT Project of Check associated

The first step is to collect all the alerts (and checks) belonging to the Job with their eventual parameters and to write them in the YAML source file inside the model folder of the DBT project, also tagging every data source for the next step. In order to do so, it was adopted snakeyaml as parsing library to write every check, considering its datasource, tables, columns and parameters in the correct location in the file.

5.4.2 Handling of multiple datasource for a single job

Since a single DBT test command can execute the tests defined over one single datasource, in the case of a job, with multiple tests belonging to different datasources, it is necessary to launch different DBT test command for every datasource. In order to do so, in the previous step, every datasource has to be marked with a tag in the YAML file.
The final command adopted is :

Listing 5.1: DBT Job command

```bash
#!/bin/bash

```

```
dbt test --select tag:target_name --store-failures --threads
    num_threads --target target_name
```

Every flag has the following explanation:

- **–select tag:tag_name** : to select a subset of checks inside the same file belonging to a datasource with that tag
- **–store-failures** : this flag indicates that DBT has to create one or more tables depending on the amount of failed tests in order to memorize the results and the rows of the failed tests. This will be useful to retrieve some statistics about the failure and the number of rows affected
- **–target target_name** : to make DBT adopt a specific datasource with their connection parameters specified in the profiles.yml file
- **–threads num_threads** : this flag indicates how many threads will be created in order to execute all the queries necessary to apply the test

5.4.3 Output parsing

After every job has finished its execution, a union of every DBT output process is performed to obtain the results.
The principal part to retrieve from the output is the test results; every test can have three different outcomes:

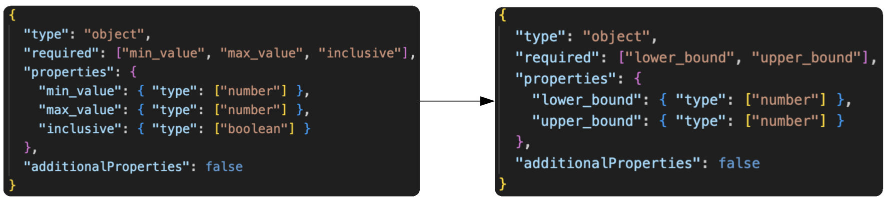
- **PASS**
- **FAIL**: in this case, the output shows also the location of the table created to store the rows that did not pass the test.
- **ERROR**: This outcome means that DBT cannot perform the query to check it, the reason is shown in the output as well.

5.5 Check conversion

In order to deeply strengthen the integration of DBT in Alertable, it has been implemented a conversion system between checks and alerts.
The conversion has been adjusted with a change in the DBT format in order to make it work as the one used in Alertable for the Apache Spark execution. The conversion has the role to transform the check and creates the corresponding alert template from the DBT format into the standard template used by Alertable since DBT jobs does not require the definition of an alert. The core part of a check template, with the exclusion of id, creation and update data informations, are:

- The name
- The outputs: a list of fields name to represent the outputs of the data quality check - like success, fail, skip or null count.
- The template: it is a json object composed by the parameters and their type.

So, the conversion process changes the DBT check type, its parameters and also creates the alert condition associated in order to be equal to the Alertable type. An example of conversion is shown below, where the **dbt-utils.accepted_range** is converted to the Alertable corresponding values_between_boundaries:



```
{
  "type": "object",
  "required": ["min_value", "max_value", "inclusive"],
  "properties": {
    "min_value": { "type": ["number"] },
    "max_value": { "type": ["number"] },
    "inclusive": { "type": ["boolean"] }
  },
  "additionalProperties": false
}
```

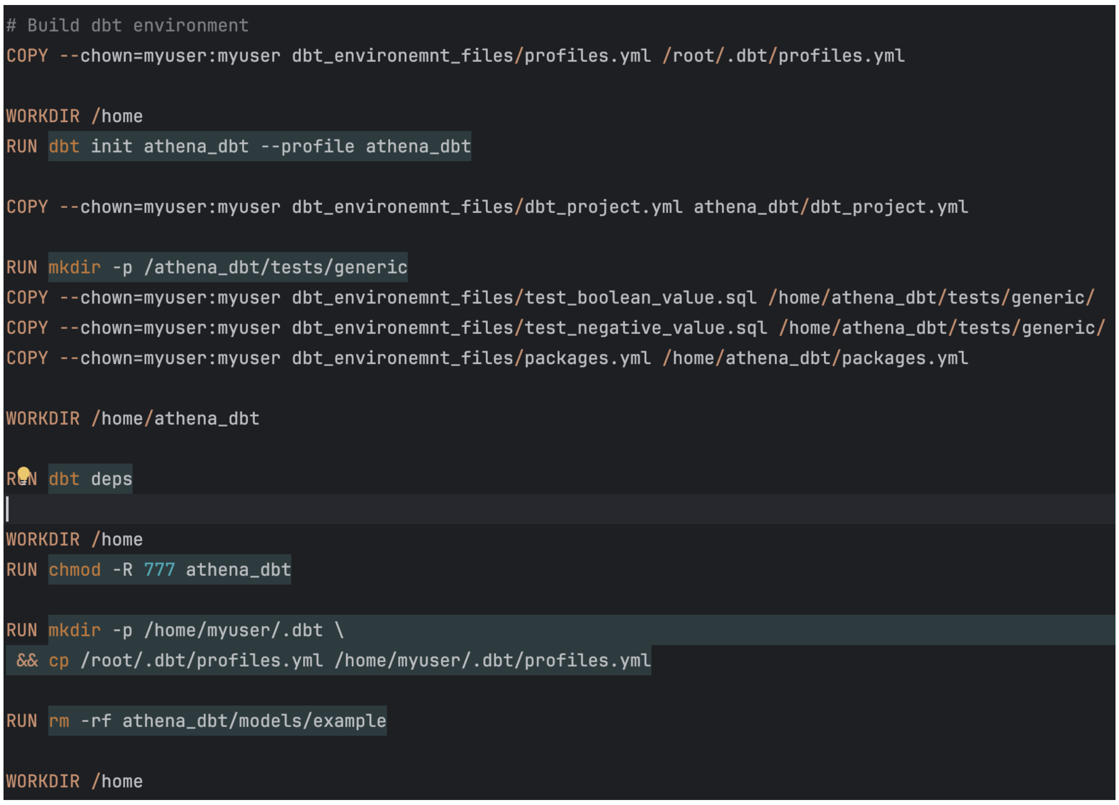
```
{
  "type": "object",
  "required": ["lower_bound", "upper_bound"],
  "properties": {
    "lower_bound": { "type": ["number"] },
    "upper_bound": { "type": ["number"] }
  },
  "additionalProperties": false
}
```

Figure 5.8: Check conversion

5.6 Deploying through Docker

Since Alertable is deployed in an EC2 instance with a docker image memorized in the AWS ECR image repository, it is necessary to extend the actual Docker image in order to prepare and initialize the DBT environment for the execution. The principal instructions added to the docker image are the following:

- **Copy of profiles.yml file**: from the source environment, the DBT profiles.yml file has to be copied in the right location of the deploy environment in order to be recognized and adopted by DBT and for different user execution
- **DBT Project inizialization**: the DBT init command is executed , the generic tests are copied in the right folder and the file packages.yml is created with the libraries adopted
- **Packages install**: the DBT deps command is executed in order to install the project dependencies



```
# Build dbt environment
COPY --chown=myuser:myuser dbt_environemnt_files/profiles.yml /root/.dbt/profiles.yml

WORKDIR /home
RUN dbt init athena_dbt --profile athena_dbt

COPY --chown=myuser:myuser dbt_environemnt_files/dbt_project.yml athena_dbt/dbt_project.yml

RUN mkdir -p /athena_dbt/tests/generic
COPY --chown=myuser:myuser dbt_environemnt_files/test_boolean_value.sql /home/athena_dbt/tests/generic/
COPY --chown=myuser:myuser dbt_environemnt_files/test_negative_value.sql /home/athena_dbt/tests/generic/
COPY --chown=myuser:myuser dbt_environemnt_files/packages.yml /home/athena_dbt/packages.yml

WORKDIR /home/athena_dbt

RUN dbt deps

WORKDIR /home
RUN chmod -R 777 athena_dbt

RUN mkdir -p /home/myuser/.dbt \
 && cp /root/.dbt/profiles.yml /home/myuser/.dbt/profiles.yml

RUN rm -rf athena_dbt/models/example

WORKDIR /home
```

Figure 5.9: DBT Dockerfile

Chapter 6

Experiments

In this chapter, there will be a comparison in terms of efficiency, scalability and integration capabilities of two differents ways to conduct the job execution with the use of Apache Spark and AWS EMR in one situation and the use of DBT in the other 5

6.1 Considerations

Before proceeding into the analysis and tests of the job execution time for both the modalities, it has to be considered each tool strength and weakness.
Apache Spark is well known for its great capabilities in managing a big volume of data and for its horizontal capabilities thanks to the cluster execution.
Furthermore, due to the ECS service offered by AWS that represents a fully managed orchestration service which exploits the cloud capabilities in terms of scalability, efficiency and easy of use, the execution with Spark grants great efficiency for heavy load and, particularly, big data warehouses.
On the other hand, DBT does not have an horizontal scalability and its executions depend solely on the datasource on which it is attached. Despite that, DBT provides a parallel execution mode with the specifications on the number of threads and also in case of multiple datasources in one single job, it naturally divides the data loads between the different datasources.
In addition to that, in case of small to medium load Spark with EMR can possibly be an overkill solution, since DBT only requires the definition of some queries or YAML file.
Another consideration is that it would be much easier to add different kind of tests in DBT, given that in most cases it does provides a library that has it instead in Spark it would be sometimes necessary to programming it.
Specifically for Apache Spark execution time, it should be taken into account that

there may be a bootstrapping phase for creating the clusters, which adds overhead.

6.2 Datasources

In this section there will be an introduction of the datasources used for the comparison:

- **Athena DB**: this datasource is composed by four different tables, listed below with the following characteristics:
  - **report_km_auto**: with 7 columns and a total of 209287 rows
  - **devices**: with 79 columns and a total of 75768 rows
  - **warehouse_and_retail_sales**: with 9 columns and a total of 307645 rows
  - **survey**: with 10 columns and a total of 32445 rows
  - **unisalute_history**: with 10 columns and a total of 418216 rows
  - **covid_data**: with 10 columns and a total of 100663 rows
- **MySQL**: this datasource uses three different tables listed below with the following characteristics:
  - **emission_factors**: with 9 columns and a total of 1016 rows
  - **mega_results**: with 4 columns and a total of 2362 rows
  - **vehicles**: with 17 columns and a total of 220225 rows
  - **border_crossing**: with 9 columns and a total of 397910 rows
  - **real_estate_sales**: with 13 columns and a total of 520000 rows
  - **feed_grains**: with 19 columns and a total of 522002 rows

6.3 Configurations

In this section are presented the machine characteristics and configurations adopted for the experiments:

6.3.1 MySQL instance

The MySql instance is created with AWS RDS in a machine with the following characteristics:

- **Instance class**: db.t4g.micro

- **vCPU**: 2
- **RAM**: 1 GB
- **Storage type**: General Purpose SSD (gp2) with 20 GB

6.3.2 Cluster EMR

The EMR cluster was configured with the following characteristics:

- **Master and core nodes Instance class**: m4.xlarge
- **vCPU**: 4 vCore
- **RAM**: 16 GB
- **Storage type**: EBS only storage with 32 GiB

In the following experiments, there will be tests of different configurations of the cluster depending on the number of core nodes (1, 2 and 4)

6.4 Execution Time Comparison

The comparison between the two modes will be done for different scenarios considering the datasources involved, how much rows are considered, specifically for DBT, how many threads are adopted for the execution and for Spark how many core nodes composed the EMR cluster

- **Case 1**: 25 checks belonging to Athena datasources **report_km_auto** and **devices** and a total of 285055 rows
- **Case 2**: 38 checks belonging to Athena datasources **report_km_auto**, **devices** and **warehouse_and_retail_sales** with a total of 592700 rows
- **Case 3**: 69 checks belonging to Athena datasources **report_km_auto**, **devices**, **warehouse_and_retail_sales**, **survey**, **unisalute_history** and **covid_data** with a total of 1144024 rows
- **Case 4**: 38 checks belonging to MySQL datasources **emission_factors**, **mega_results** and **vehicles** with a total of 223603 rows
- **Case 5**: 53 checks belonging to MySQL datasources **emission_factors**, **mega_results**, **vehicles** and **border_crossing** with a total of 621513 rows

- **Case 6**: 69 checks belonging to MySQL datasources **emission_factors**, **mega_results**, **vehicles**, **border_crossing**, **real_estate_sales** and **feed_grains** with a total of 1663515 rows

- **Case 7**: 63 checks with the datasources of **Case 1** and **Case 4** with a total of 508658 rows:

- **Case 8**: 90 checks with the datasources of **Case 2** and **Case 5** with a total of 1214213 rows

- **Case 9**: 138 checks with the datasources of **Case 3** and **Case 6** with a total of 2807539 rows

Case	Checks	Rows	DBT (s)	Spark - 1 core node (s)	Spark - 2 core node (s)	Spark - 4 core node (s)
1	25	285,055	207	313	280	269
2	37	592,700	222	424	401	370
3	69	1,144,024	243	723	641	607
4	38	223,603	12	383	353	329
5	53	621,513	17	384	356	328
6	69	1,663,515	46	400	357	338
7	63	508,658	208	641	576	545
8	90	1,214,213	218	791	682	658
9	138	2,807,539	249	1118	984	917

Table 6.1: Execution Time Comparison for Different Cases

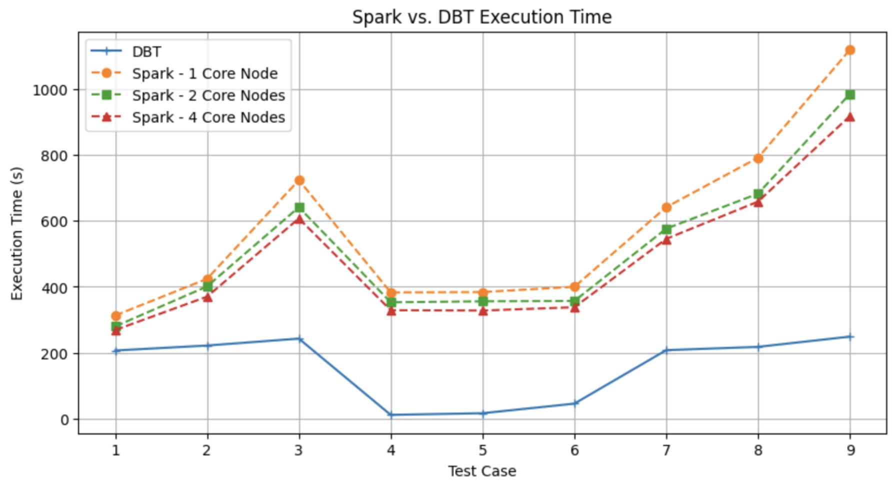


Figure 6.1: DBT vs Spark Execution Time

6.5 Final Considerations

As shown in the above comparison, DBT jobs outperform Spark in terms of pure execution speed, particularly when the datasource is MySQL.
Since Athena queries operate directly on files stored in Amazon S3, query execution in this case relies on reading raw CSV files, which is inherently slower than querying structured and indexed databases.
Athena performs significantly better when working with columnar storage formats like Parquet, as they allow a more efficient data retrieval and compression.

Additionally, despite the Spark execution was performed with a different number of core nodes, the gap is still significantly large; in order to have comparable execution time, the cluster in this case have to be composed by at least 8 core nodes in some cases if not 16 or more expecially for the last cases.
It has also to be considered that the DBT performances are tightly coupled to the underlying database infrastructure, since DBT essentially translates the check controls into SQL queries, leveraging the processing power of the database engine. This means that if the database server is under heavy load or has limited resources, the execution time can be negatively impacted. On the other hand, Spark scales horizontally, distributing computations across multiple nodes, enabling it to handle large scale transformations and check more efficiently than a single node database systems, decoupling more effectively the database usage from the job requirements.
From a cost perspective, DBT has the advantage of not requiring additional infrastructure like the EMR cluster saw in the Spark case.
Ultimately, DBT has the advantage of being easier to extend and to modify since it can counts on more library specifically designed for data quality, requiring the modifications of just YAML files and the writing of SQL queries.

Chapter 7

Conclusions

The growing importance of data quality in modern organizations has underscored the need for tools that can ensure the accuracy, reliability and observability of data.
This thesis focused on addressing these challenges by integrating DBT, a robust framework for data transformation and testing, into the existing Alertable platform, thereby extending its capabilities in data quality management.

This enhancement has turned Alertable into a more versatile and powerful tool for monitoring and ensuring data integrity. Specifically, the system's ability to execute DBT checks on diverse datasources, such as AWS Athena and MySQL, has significantly broadened its applicability in enterprise data ecosystems.
Generally speaking, with really heavy size of data Spark has the advantage of horizontally scaling naturally the process, however for small-medium to light heavy load of data, DBT has proven its adaptability and fastness with respect to Apache Spark.

The experimental evaluation of the DBT job, despite the execution with Spark provides more advanced types of checks, like the applying of a pre filter on a table before performing the checks, the setting of complex threshold, conditions in order to trigger an alert and a more granular control on the job parameters, has shown its potential.
With future enhancement in order to fully align the functionality of the actual Alertable engine and the execution with DBT and with optimization in terms of overhead reduction, the job execution with DBT can constitute a valid alternative, capable of diminuishing the cost, since it only requires an existing datasource to perform the checks and does not add additionally computational resource, like a cluster.

Also, it can be added a new kind of tests to enrich the actual possible checks, due to the presence of libraries like dbt-expectations or dbt-utils.

By focusing on data quality as the cornerstone of reliable decision-making, this thesis provides a robust framework for extending Alertable's capabilities.
The integration of DBT does not only strengthens its position as a tool for data validation but also demonstrates the potential for combining modern data engineering practices with observability platforms to meet the increasing complexity of data ecosystems.

Bibliography

[1] IBM. *What is data quality?* 2024. URL: https://www.ibm.com/topics/data-quality#:~:text=the%20next%20step-,What%20is%20data%20quality%3F,governance%20initiatives%20within%20an%20organization. (cit. on p. 3).

[2] Bill$_I$*nmon. Data Observability, The Reality.* The Ravit Show, 2023 (cit. on p. 4).

[3] GreatExpectations. *Great Expectations overview.* 2024. URL: https://docs.greatexpectations.io/docs/core/introduction/gx_overview (cit. on p. 9).

[4] GreatExpectations. *Great Expectations validation.* 2024. URL: https://docs.greatexpectations.io/docs/core/define_expectations/test_an_expectation?procedure=instructions (cit. on p. 11).

[5] GreatExpectations. *Great Expectations overview.* 2024. URL: https://legacy.017.docs.greatexpectations.io/docs/0.14.13/reference/data_docs/ (cit. on p. 13).

[6] GreatExpectations. *Great Expectations Batch Definition.* 2024. URL: https://docs.greatexpectations.io/docs/core/define_expectations/retrieve_a_batch_of_test_data?procedure=sample_code (cit. on p. 14).

[7] SodaCore. *Soda Core checks.* 2024. URL: https://docs.soda.io/soda-cl/soda-cl-overview.html (cit. on p. 15).

[8] Vaadin. *Vaadin.* 2024. URL: https://vaadin.com/benefits (cit. on p. 18).

[9] Spring. *Spring Boot.* 2024. URL: https://spring.io/why-spring (cit. on p. 19).

[10] AWS EMR. *Aws EMR.* 2024. URL: https://aws.amazon.com/it/emr/ (cit. on p. 19).

[11] getdbt. *Analytical Engineering Role.* 2024. URL: https://www.getdbt.com/blog/analytics-engineer-vs-data-analyst (cit. on p. 29).

[12] getdbt. *Analytical Engineering comparison*. 2024. URL: `https://www.getdbt.com/what-is-analytics-engineering` (cit. on p. 30).

[13] hevodata. *Dbt Use Cases*. 2024. URL: `https://hevodata.com/learn/dbt-use-cases/` (cit. on p. 30).

[14] Dbt. *Dbt Commands*. 2024. URL: `https://docs.getdbt.com/reference/dbt-commands` (cit. on p. 31).

[15] AWS Amazon. *Aws Ec2*. 2024. URL: `https://aws.amazon.com/it/ec2/features/` (cit. on p. 41).

[16] AWS Amazon. *Aws Ecr*. 2024. URL: `https://aws.amazon.com/it/ecr/` (cit. on p. 42).

[17] AWS Amazon. *Aws Athena*. 2024. URL: `https://aws.amazon.com/it/athena/` (cit. on p. 42).

[18] AWS Amazon. *Aws RDS*. 2024. URL: `https://aws.amazon.com/it/rds/` (cit. on p. 42).

[19] AWS S3. *Aws s3*. 2024. URL: `https://aws.amazon.com/it/s3/` (cit. on p. 42).

[20] Docker. *Docker*. 2024. URL: `https://docs.docker.com/get-started/docker-overview/` (cit. on p. 43).

Acronyms

AI	artificial intelligence
AWS	Amazon Web Services
DBT	Data Build Tool
RDS	Relational Database Service
DQ	Data Quality
EMR	Elastic Map Reduce