Motor de reglas de negocio para computación distribuida en la nube
Business rule engine for distributed computing on the cloud

Trabajo de Fin de Grado
Curso 2024–2025

Autor
Gina Andrea Cassin

Director
Eduardo Huedo Cuesta

Calificación: 10 (Matrícula de Honor)

Grado en Ingeniería Informática
Facultad de Informática
Universidad Complutense de Madrid

Motor de reglas de negocio para computación distribuida en la nube
Business rule engine for distributed computing on the cloud

Trabajo de Fin de Grado en Ingeniería Informática

Autor
Gina Andrea Cassin

Director
Eduardo Huedo Cuesta

Convocatoria: *Junio 2025*

Calificación: 10 (Matrícula de Honor)

Grado en Ingeniería Informática
Facultad de Informática
Universidad Complutense de Madrid

19 de junio de 2025

Dedicatoria

A mis amigos y familia, que me han acompañado y me han apoyado a un océano de distancia. En particular, a Cesar, que tu constancia muchas veces me ha servido de ejemplo.
A toda la gente que he conocido en esta última etapa de mi vida, y de alguna forma u otra, me han apoyado.
Y sobre todo, a Santi, quien ha estado siempre ahí a mi lado, quien no ha dubitado ni un segundo en compartir esta loca aventura.

Resumen

Motor de reglas de negocio para computación distribuida en la nube

Ante el creciente volumen de datos generados diariamente por las organizaciones y su necesidad de utilizarlos para la toma de decisiones, se ha vuelto cada vez más relevante la capacidad de definir y aplicar reglas de negocio a gran escala. Si bien Apache Spark se ha consolidado como una plataforma líder para el procesamiento distribuido de datos, existe una carencia de herramientas de código abierto y de fácil uso que permitan integrar la ejecución de reglas de negocio directamente en sus flujos de trabajo. Este proyecto presenta *Stickler*, un motor de reglas de negocio de código abierto diseñado específicamente para computación distribuida con Spark, orientado a la integración en entornos de computación en la nube.

Stickler está concebido para aplicar una lógica estricta basada en reglas, manteniendo a su vez la flexibilidad necesaria para adaptarse ágilmente a estrategias comerciales constantemente cambiantes. El motor es dinámico, y cuenta con definiciones configurables, ofreciendo una amplia variedad de opciones de ejecución para satisfacer necesidades comerciales específicas.

Con un enfoque en la escalabilidad, el rendimiento y la transparencia, *Stickler* también incorpora un sistema de seguimiento de historial para garantizar visibilidad y trazabilidad.

Se asume que el lector está familiarizado con conceptos básicos de programación y de computación distribuida para poder comprender adecuadamente el contenido de este documento.

Palabras clave

Motor de reglas de negocio, Computación distribuida, Apache Spark, PySpark, Big Data, Computación en la nube, Código abierto.

Abstract

Business rule engine for distributed computing on the cloud

As organizations generate vast amounts of data daily and seek to leverage it for informed decision-making, the ability to define and apply business rules at scale has become increasingly fundamental. Although Apache Spark is widely used for distributed data processing, there is a lack of open-source, user-friendly tools that integrate business rule execution directly into Spark-based workflows. This project introduces *Stickler*, an open-source business rule engine designed specifically for distributed computing using Spark, with a focus on integration into cloud-based environments.

Stickler is built to enforce strict rule-based logic while maintaining the flexibility to adapt to constantly changing business strategies. The engine is highly dynamic, driven by configurable rule definitions, and offers a wide range of execution options, giving users control over how rules are applied to meet specific business needs.

Designed with scalability, performance, and transparency in mind, *Stickler* also includes a built-in history-tracking system to ensure visibility and accountability.

It is assumed that the reader has a basic understanding of programming and distributed computing concepts to engage fully with the content of this document.

Keywords

Business Rule Engine, Distributed computing, Apache Spark, PySpark, Big Data, Cloud computing, Open-source.

Contents

1. Introduction **1**
- 1.1. Motivation 1
- 1.2. Objectives 3
- 1.3. Work Plan 4

2. State of the Art **7**
- 2.1. BRE vs BRMS vs DMS 7
- 2.2. Applications in Industry 9
- 2.3. Operational Workflow of a BRE 9
  - 2.3.1. Basic rule definitions 10
  - 2.3.2. BRE Operational Workflow 10
- 2.4. Rule Representation in BREs 11
  - 2.4.1. If-then Rules 11
  - 2.4.2. Decision Tables 12
- 2.5. Rule Evaluation in BREs (Inference) 13
  - 2.5.1. Forward Chaining 13
  - 2.5.2. Backward Chaining 13
  - 2.5.3. Sequential Approach 14
- 2.6. Hit Policies in BREs 15
- 2.7. BREs in Distributed Computing 15
  - 2.7.1. Cloud Services 16
- 2.8. BREs and Big Data 17
  - 2.8.1. The Four V's of Big Data 17
  - 2.8.2. How Inference and Rule Application Differ Between Traditional (Legacy) and Data-Intensive Rule Engines 17
  - 2.8.3. Efficiency in Execution of Business Rules in Distributed Environments 18
  - 2.8.4. Limitations of Current Open-Source Rule Engines in Big Data Processing 21
- 2.9. Existing Business Rule Engine Solutions 22
  - 2.9.1. Table Summary 23

3. System Design and Architecture **27**
3.1. Problem Definition . . . 27
3.2. Requirements and Design Considerations . . . 28
3.2.1. Functional Requirements . . . 28
3.2.2. Non-Functional Requirements . . . 29
3.2.3. Trade-offs: Inference Complexity vs. Large-Scale Execution . . 30
3.3. Architecture Overview . . . 30
3.3.1. Design Constraints in Rule Logic . . . 30
3.3.2. Rule Execution Types . . . 31
3.3.3. Input and Output . . . 32
3.3.4. Rule Execution Process . . . 36

4. Implementation **39**
4.1. Technology Stack and Tooling . . . 39
4.2. Architecture . . . 42
4.3. Core Components of the Rule Engine . . . 45
4.3.1. Rule Parser and Validation . . . 47
4.3.2. Rule Application Strategies . . . 48
4.3.3. Execution Order, Dependencies and Cascading Rule Effects . 50
4.3.4. History-Tracking Mechanism Implementation . . . 51
4.4. Anatomy of the Implementation . . . 52
4.5. Testing and Error Handling . . . 55
4.5.1. Unit Testing . . . 56
4.5.2. Integration Testing . . . 59
4.5.3. Error Handling and Logging . . . 60

5. Cloud Deployment and Performance **63**
5.1. Scenario Definition . . . 63
5.2. Dataset Description and Preparation . . . 63
5.3. Performance Evaluation Metrics . . . 66
5.4. Rule Definition . . . 67
5.5. Set up . . . 70
5.6. Results . . . 70
5.6.1. Execution time . . . 70
5.6.2. Resource usage . . . 71
5.6.3. Cost . . . 73
5.6.4. Accuracy . . . 75

6. Conclusions and Future Work **79**
6.1. Conclusions . . . 79
6.2. Future Work . . . 80

Bibliography **83**

A. JSON input example **89**

B. AWS Glue Script code **93**

C. User Guide **95**

List of figures

1.1. Abstract factory pattern representation. By Refactoring Guru [55]. . . 2

2.1. Discounts application diagram. 15
2.2. IBM ODM Spark integration. From decisions-on-spark, on GitHub [49]. 20

3.1. Simplified rule engine execution flow. 37

4.1. Technology stack and frameworks. 40
4.2. Architecture diagram. 43
4.3. Core components of the Rule Engine. 46
4.4. Rule validation sequence diagram. 48
4.5. Rule application strategies diagram. 49
4.6. Discounts application diagram. Cascade group 0: Pink. Cascade group 1: Green. 50
4.7. UML class diagram. 52
4.8. Sequence diagram. 54
4.9. Chain responsability pattern. By Refactoring Guru [56]. 55

5.1. Comparison of execution time for different AWS Glue worker configurations. 71
5.2. CPU load for the first configuration (10 workers). 72
5.3. Memory profile for the first configuration (10 workers). 72
5.4. Comparison of execution cost for different AWS Glue Worker configurations. 75

C.1. Overview of Stickler's operational workflow. 95

List of tables

2.1. Membership type decision table. Inspired by the decision table example found in Camunda (2024) [21]. 12
2.2. BREs comparison table. 24
2.3. BREs comparison table, continuation. 25

3.1. Original input DataFrame. 36
3.2. Output DataFrame after applying `discountPrices` rule. 36
3.3. History DataFrame capturing original values and rule-generated modifications. 36

4.1. Example History DataFrame for the `discountPrices` rule (same as Table 3.3). 51
4.2. Unit tests. 57

5.1. Summary of Rules for Performance Evaluation. 67
5.2. Flight row with applied rules and final fare. 76
5.3. Flight row where *HighDemandSurcharge* applied and *OffPeakDayDiscount* was blocked. 77
5.4. Flight row where *ExtendedLayoverDiscount* applied. 77

List of acronyms

Amazon EMR *Amazon Elastic MapReduce*

API .... *Application Programming Interface*

AWS .... *Amazon Web Services*

BRE .... *Business Rule Engine*

BRMS .... *Business Rules Management System*

CSV .... *Comma Separated Values*

DMN .... *Decision Model and Notation*

DMS .... *Decision Management System*

DPUs .... *Data Processing Units*

ETL .... *Extract, Transform, and Load*

HDFS .... *Hadoop Distributed File System*

JSON .... *JavaScript Object Notation*

ML .... *Machine Learning*

IBM's ODM .. *Operational Decision Manager*

PyPI .... *Python Package Index*

RDBMS ...... *Relational Database Management System*

RDD .... *Resilient Distributed Dataset*

SQL .... *Structured Query Language*

UDFs .... *User-Defined Functions*

Chapter 1

Introduction

In today's world, companies rely heavily on data to make informed decisions, automate processes, and improve efficiency. As businesses scale, the amount of data they need to process grows rapidly, and handling large datasets becomes a challenge. Many companies turn to distributed computing frameworks like Apache Spark to process data at scale, especially when working in cloud environments. However, one area where existing tools fall short is the execution of business rules within these distributed systems.

Business rule engines allow companies to define and enforce logic, such as applying specific conditions to data, filtering records, or modifying values based on predefined rules. These engines are commonly used in fields like finance, healthcare, and retail, where decisions must be automated at a large scale. While several rule engines are available, most are either proprietary, designed for small-scale applications, or not optimized for distributed environments. More importantly, there is no widely adopted open-source BRE (*Business Rule Engine*) built explicitly for Spark, a framework broadly used in data engineering.

This project aims to fill that gap by developing a business rule engine named *Stickler*, for distributed computing on the cloud. The name *Stickler* was chosen to reflect the engine's core philosophy of strict adherence to business rules. In common usage, a *stickler* refers to someone who believes in closely following rules or in maintaining a high standard of behavior [7].

The goal is to create a system that allows users to define business rules in a structured way and apply them efficiently to large datasets using Spark on a Python codebase, specifically via PySpark. The engine will be designed to be flexible, allowing dynamic rule execution while maintaining performance. Additionally, it will include features for tracking rule execution history, ensuring transparency in how data is modified over time.

1.1. Motivation

In many industries, business rules are at the core of decision-making. Financial institutions use rules to detect fraud or approve loans based on risk factors. Health-

care organizations apply rules to process insurance claims and ensure compliance with regulations. Retail companies rely on business rules for dynamic pricing, personalized customer recommendations or discount applications. In all these cases, business rules must be implemented efficiently to massive datasets, sometimes in real-time or near real-time.

This project actually originated from a real-world business need within a retail company: the requirement for dynamic pricing. Dynamic pricing refers to the practice of adjusting prices in real-time based on factors like demand, inventory levels, competitor pricing, or market conditions. This type of pricing allows businesses to maximize their revenue by continuously adapting their prices to reflect the current market dynamics [30].

The first approach of the solution involved rules being hardcoded directly into the application. This meant that whenever a change in the pricing logic was required, such as adjusting a parameter, or even adding new conditions, it implied a complete cycle of code development, testing and redeployment. As the application grew, this hardcoded approach became an obstacle to agility.

To address this challenge, a refactoring effort was initiated, implementing a rule engine with an abstract factory pattern. The abstract factory pattern is a creational design pattern that provides an interface for creating families of related objects without specifying their concrete classes [55]. In this context, it was used to decouple the creation of rule components from their usage, allowing different families of rule implementations to be produced based on contextual needs.

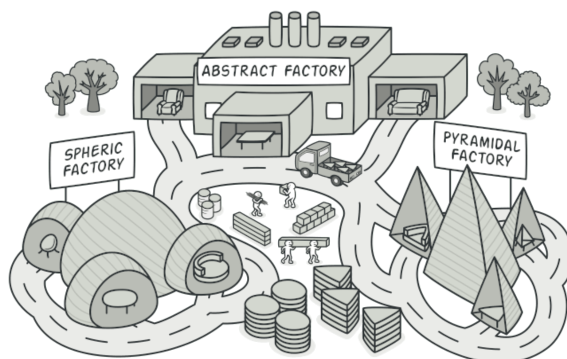

Figure 1.1: Abstract factory pattern representation.
By Refactoring Guru [55].

While this initially provided increased flexibility, it introduced a new set of challenges. Specifically, each aspect of a rule required its own specific implementation, leading to a substantial expansion of the number of classes and resulting in a more complex and tightly coupled codebase. Furthermore, this solution was tailored exclusively to dynamic pricing requirements; it proved unsuitable for implementation in other projects with similar needs but differing contextual demands.

As a consequence, this last approach to implementing business rules became a bottleneck, unable to adapt to the growing and constantly changing pricing strategies required by the business or extend to any other context. This is when the need for a more versatile and scalable solution became clear, leading to the design and

development of the *Stickler* engine, since there were no existing solutions that could be easily adapted to the specific requirements of the company.

This project is significant because it bridges the gap between business rule processing and scalable distributed computing on the cloud. By taking advantage of Spark, it provides a flexible and efficient solution that integrates rule evaluation into large-scale data workflows.

Another important aspect of this project is transparency. Data-driven decision-making requires trust in how data is processed. If a value in a dataset is modified due to a rule, organizations need a clear record of what rule was applied and why. This engine will include a built-in mechanism to track rule execution history, providing full visibility into how data evolves over time. This feature is especially useful for compliance-heavy countries, where entities must adhere to strict regulations regarding data.

1.2. Objectives

The primary objective of this project is to design and implement an open-source, Spark-native business rule engine that easily integrates with distributed cloud computing environments. The key goals include:

- **Scalability**: The engine must be able to handle large datasets efficiently by using Spark's distributed computing capabilities to process data in parallel.
- **Universal applicability**: The system must integrate into any project type and scope.
- **Execution control**: It should support different execution types, allowing rules to be applied in such a way that they are completely aligned to the requirements of the business.
- **History tracking**: A mechanism should be built to record and document all data alterations performed during rule execution, providing full transparency and traceability.
- **Performance optimization**: The implementation should minimize expensive operations, such as data shuffling, while ensuring that rule execution does not significantly slow down data processing.
- **User-friendliness**: The engine should be designed in a way that allows non-expert users to define rules using simple expressions while still supporting complex logic for advanced use cases from more experienced users.
- **Integration with cloud environments**: Since many organizations use cloud-based data lakes and analytics platforms, the engine should be cloud-friendly, integrating well with existing distributed cloud data pipelines.

1.3. Work Plan

The development of this project will follow a structured approach to ensure an efficient implementation. The work plan is divided into several key phases, each addressing different aspects of the project:

1. **Research**

   The initial phase involves understanding the problem domain, analyzing existing rule engines, and identifying their limitations, particularly in the context of Spark and cloud computing.

2. **System design and architecture**

   This stage will define the functional and non-functional requirements of the system, ensuring that the proposed solution meets the goals. Once the requirements are established, the next step is designing the architecture of the rule engine. This also includes defining how rules will be defined and structured, how they will be applied to datasets, and how the system will handle rule execution efficiently. Special attention will be given to scalability and ease of integration with existing data pipelines.

   In this part, the expected behavior of the system will be defined with detail, including the expected input and output formats.

3. **Implementation**

   The core development phase will focus on building the rule engine using PySpark. This will involve:

   - Implementing a rule parser and execution framework.
   - Ensuring an accurate rule application, adhering to defined specifications.
   - Managing complex rule execution sequences, dependencies, and cascading effects.
   - Developing a history-tracking mechanism to maintain a clear record of data transformations.

   The rule engine must be designed to be flexible and ease of deployment across different environments, specially in cloud-based platforms.

4. **Testing and validation**

   To ensure the correctness and efficiency of the rule engine, a series of small tests will be conducted:

   - **Unit tests** to validate individual components.
   - **Integration tests** to validate the overall functionality of the rule engine using a representative example.

5. **Cloud deployment and performance evaluation**

   This phase is dedicated to assess the rule engine's performance in a realistic operational context. The evaluation will utilize a flight prices dataset to simulate a dynamic pricing scenario, such as a large-scale commercial event like Black Friday. This scenario will involve applying a complex set of rules to recalculate flight prices based on various factors, including flight routes and promotional discounts.

   The cloud deployment will be performed in AWS (*Amazon Web Services*).

   The performance evaluation will focus on the following key aspects:

   - **Scenario definition:** Designing a comprehensive set of rules that simulate dynamic pricing strategies during a commercial event.
   - **Metric measurement:**
     - **Execution speed:** Measuring the total time taken to process the entire dataset and apply all defined rules.
     - **Resource utilization:** Monitoring CPU load and memory during rule execution to identify potential bottlenecks and ensure efficient use of cloud resources.
     - **Accuracy verification:** Validating the correctness of recalculated prices against expected results, aligned with the "accurate rule application" goal from the implementation phase.
     - **Cost:** Evaluating the cost of executing rules in a cloud environment, particularly when processing a large dataset.

6. **User guide**

   A comprehensive user guide will be created, represented in a well-documented README published on the project's public GitHub repository.

7. **Final evaluation and future work**

   The final phase will involve evaluating the success of the project, comparing its performance against initial expectations and identifying potential areas for future enhancements.

Chapter 2

State of the Art

Business Rule Engines (*BREs*) are software systems designed to automate decision-making by evaluating and executing predefined business rules. These rules define logic in a specific, structured format, allowing organizations to separate decision-making from application code. This modularity offers significant advantages. It empowers developers, and in some cases, even non-technical users to define rules independently, without altering the core software, ensuring autonomy and flexibility for rapid adaptation to constant changing business needs.

In recent years, the shift towards cloud computing and large-scale data processing has posed new challenges for traditional rule engines. Many legacy systems were designed for single-node execution and struggle to scale efficiently when applied to massive datasets. For this reason, modern solutions have emerged, using distributed computing frameworks to handle rule execution across large data volumes, but none of them leverage Spark directly, which is the industry standard. Apache Spark currently is being utilized by more than 80% of Fortune 500 companies for big data processing [25].

2.1. BRE vs BRMS vs DMS

A **Business Rule Engine (BRE)** is fundamentally a software component or library designed for the primary purpose of executing predefined business rules [29]. Its core function is to evaluate specific conditions based on input data and trigger corresponding actions or decisions [43]. Their objective is to decouple the business logic from the main application code, allowing rules to be managed externally without requiring code redeployment. BREs are typically embeddable components within larger systems, focusing on efficient rule execution [29].

Business Rule Management Systems (BRMS) represent an evolution from the standalone BRE, providing a comprehensive platform for the entire lifecycle of business rules. It essentially packages a BRE with extensive management capabilities [29]. Key features include a centralized repository for storing and organizing rules, version control [29], user-friendly authoring environments (often graphical or using controlled natural language) accessible to business analysts, testing and simulation

tools, deployment management, and governance features like access control [11].

A **Decision Management Suite (DMS)** (sometimes called Digital Decisioning Platforms or DDPs by analysts like *Forrester* [24]) represent a further evolution, adopting a more holistic, decision-centric perspective [59]. DMSs aim to automate, manage, and optimize complex operational decisions by integrating business rules technology with advanced analytics (including predictive and machine learning models), data processing capabilities, and potentially optimization algorithms [24]. A key differentiator is the emphasis on explicit decision modeling, often using standards like the DMN (*Decision Model and Notation*) [46], to structure and understand complex decision logic separately from application code [59]. DMSs facilitate the integration of various inputs, including rules, predictive models, real-time data, and complex calculations, to support data-driven decision-making and enhance business agility [24].

The fundamental difference between the three lies in scope and corresponding feature sets. A BRE offers a narrow scope on the execution of predefined rules, primarily serving to decouple logic from code for performance optimization. A BRMS (*Business Rules Management System*) expands this scope to encompass the complete rule lifecycle. A DMS (*Decision Management System*) adopts the broadest scope, focusing on optimizing business outcomes by integrating analytics and models within a structured decision model framework.

The intended users also differ significantly. BREs are primarily tools for developers, who need the technical skills to configure and interact with the engine via its API (*Application Programming Interface*) or proprietary language. BRMS platforms aim to bridge the gap between IT and business. While developers are still needed for integration and potentially complex rule implementation, BRMS provide interfaces (like decision tables or graphical editors) designed for business analysts or subject-matter experts to author, test, and manage many of the rules themselves [29]. DMS platforms often involve a broader range of roles, including business analysts (for decision modeling and rule authoring), data scientists (for developing and integrating predictive models), and developers (for integration and complex logic implementation) [24] [59]. This requires a wider skill set within the team, from decision modeling principles and potentially analytics or ML (*Machine Learning*) concepts.

In terms of complexity, BREs are generally the simplest, often being lightweight libraries [29]. Their scalability depends heavily on the engine's architecture and how it is deployed within the host application. BRMS platforms are inherently more complex due to their extensive management features. However, they are typically designed for enterprise scalability and are widely used [59]. DMSs represent the highest level of complexity, integrating multiple more features, such as analytics and machine learning models. Choosing between these technologies involves a clear trade-off: the richer feature sets and broader scope of BRMSs and DMSs come at the cost of increased implementation, operational, and potentially licensing complexity [29].

2.2. Applications in Industry

Complex decision-making processes across various industries have been radically changed by Business Rule Engines (BREs). Their impact is particularly noticeable in sectors such as finance, healthcare, logistics and retail, where automated rule-based systems simplify operations, reduce human intervention, while improving accuracy.

In the financial sector, BREs play a crucial role in risk assessment, regulatory compliance, and fraud detection. Banks and lending institutions rely on rule-based decisioning systems to evaluate loan applications, assessing factors such as credit scores or income [62]. Moreover, BREs enhance fraud detection by identifying suspicious transactions based on predefined rules. Transactions exceeding certain thresholds, or unusual spending behaviors, can trigger alerts, enabling financial institutions to take proactive measures to prevent fraud [42].

In healthcare, the use of BREs is becoming increasingly common in clinical decision support systems and insurance claims processing. These engines help medical professionals make data-driven decisions by evaluating patient data against predefined medical guidelines. For example, a BRE can assist in prescribing treatments by cross-referencing a patient's symptoms, medical history, and test results with established medical protocols, ensuring more accurate and standardized care [12].

The logistics industry also benefits significantly from BREs, particularly in optimizing supply chain operations and compliance management. Companies use rule engines to automate inventory control, order fulfillment, and route planning, reducing delays and improving cost efficiency. For instance, a logistics company may implement a rule engine to prioritize shipments based on urgency and weather conditions [44].

In retail, dynamic pricing has become a common strategy for adjusting product prices in response to factors like demand, stock levels, and competitor behavior. BREs help automate these pricing decisions through clear defined rules, allowing companies to react quickly to market changes without manual intervention. For example, Deloitte has worked on dynamic pricing solutions that support large scale retailers in adjusting prices daily across thousands of items, obtaining a balance between profitability and inventory turnover [20]. In another case, a large retail chain implemented a rule-based system that automatically adapted prices in physical stores based on real-time competitor pricing and internal pricing models. The result was a measurable increase in gross margins, showing how BREs can support more flexible and effective pricing strategies at scale [38].

2.3. Operational Workflow of a BRE

The execution process within a BRE involves a structured workflow that encompasses rule definition, evaluation and execution. The definitions described in this section are inspired by foundational concepts from the *Rete algorithm*, as introduced by *Charles L. Forgy* in his paper [23]. As *Forgy* articulated, the *Rete algorithm* is a "rule-matching algorithm that is designed to achieve efficiency in matching a large number of patterns to a large number of objects".

These principles provide a basis for understanding how rules are processed efficiently in advanced rule engines, usually implemented in BRMSs such as Drools [54].

2.3.1. Basic rule definitions

Two fundamental concepts are essential to understand the operational workflow of a BRE:

- **Rule:** A most basic representation of a rule is defined as:

  IF condition THEN action

  This will be explained in more detail in the next section (Section 2.4).

  For example, consider the rule:

  IF order_weight > 5 AND delivery_zone = 'international'
  THEN shipping_cost = 25

  Here, the condition is a logical expression that evaluates to true or false [36].

- **Rule set:** A rule set, denoted as R, is a collection of individual rules:

  $$R = \{R_1, R_2, ..., R_n\}$$

  where R_i represents a rule within the set, and n indicates the total number of rules in R.

2.3.2. BRE Operational Workflow

The operational workflow of a BRE can be summarized in a series of steps [61], as detailed below:

1. **Data input:** The engine receives input data relevant to the decision being made. This data might come from user input, databases or external systems.

2. **Data parsing:** The input data is parsed and transformed into a format the engine understands, often represented as "facts" or objects in the engine's working memory. In the previous example, `order_weight` is a fact. Thus, conditions can be defined as a set of functions, applied over facts, that return *true* or *false*.

3. **Rule execution cycle:** Rules are processed using an inference engine (a concept that will be introduced in Section 2.5), taking a series of steps which are repeated until no more rules are triggered:

a) **Rule evaluation function:** Each rule R_i can be seen as a boolean function of conditions C:

$$R_i(C) \to A$$

Where C is the set of conditions and A is the resulting action. The engine compares the conditions in its working memory (the *"IF"* part), and stores a set of conditions that are satisfied by the current facts. If a fact is added or modified, the engine re-evaluates the conditions to check if any of them can be satisfied. If a fact is removed, the engine removes all conditions that are no longer satisfied. Basically, the engine reacts to changes in working memory.

b) **Conflict resolution strategy:** If multiple rules are eligible to fire, a conflict resolution strategy is applied to select which rule(s) to execute, such as *first match* or *collect*. These strategies will be explained in more detail in Section 2.6.

c) **Rule execution (Action):** The actions (the *"THEN"* part) of the selected rule(s) are executed. Actions might involve modifying facts in the working memory, invoking external services, generating alerts, updating databases or making a final decision.

4. **Output generation:** The engine produces an output, which could be the final decision, updated data, or signals for subsequent actions in a larger business process.

From this we can infer that the rule engine operates as a transformation function, mapping an input fact set to an output fact set. This relationship can be expressed as:

$$\text{BRE}(F) \to F'$$

where F represents the initial fact set provided as input, and F' denotes the resulting fact set after the engine's rule execution process.

2.4. Rule Representation in BREs

As their name indicate, Business Rule Engines (BREs) are designed to process rules. A rule typically consists of a combination of one or more conditions and one or more actions [34]. This separation ensures that business decisions can be updated independently, allowing for rapid adaptation to changing requirements [14].

These rules can be expressed in two formats: **if-then rules** and **decision tables**, both of which will be elaborated upon next.

2.4.1. If-then Rules

The most common representation of rules in BREs is the "if-then" format. These define which specific actions to be taken when certain conditions are met [34].

- **Conditions**: These are the "*if*" part of the rule, specifying the criteria that must be satisfied for the rule to be triggered. Conditions can involve comparisons, logical operations, and pattern matching against data.
- **Actions**: These are the "*then*" part of the rule, specifying the actions to be executed when the conditions are met. Actions can include modifying data, triggering other rules, sending notifications, or invoking external services. Optionally, a rule can also include an "*else*" part, which defines the actions to be taken when the specified conditions are not met. This allows for more precise control of logic execution, especially when combined with variable definitions in the rule's setup. In such cases, the else actions are triggered only if the variables are properly defined, but the main condition evaluates to false.

For example, a simple rule might be: "If a customer's order total exceeds $100, then apply a 10% discount." In this case, the condition is "order total exceeds $100", and the action is "apply a 10% discount".

2.4.2. Decision Tables

Decision tables are another representation of rules in BREs. They provide a compact and easy-to-read format for representing business rules. A decision table consists of a grid where each row represents a specific rule, and each column represents a condition or an action. The cells of the table contain the values or expressions that define the conditions and actions for each rule.

This format is particularly useful when there are multiple conditions and actions that are similar, enabling the identification of inconsistencies such as overlapping or missing cases.

The decision engine processes these rules sequentially, executing them row by row from the table's beginning to its end [33].

To illustrate, Table 2.1 presents a decision table for a membership system in an e-commerce platform, where spending amount, loyalty points and account age serve as the conditions to determine the corresponding membership type, which constitutes the action.

Spending (EUR)	**Loyalty points**	**Account age (months)**	**Membership type**
≥ 1000	≥ 500	≥ 12	Platinum
≥ 500	≥ 250	≥ 6	Gold
≥ 100	≥ 50	Any	Silver
< 100	Any	Any	Bronze

Table 2.1: Membership type decision table. Inspired by the decision table example found in Camunda (2024) [21].

To demonstrate the table's functionality, a fictional customer, *Customer A* has spent 600 EUR, has accumulated 300 loyalty points, and has been a member for 8 months.

The rule engine, during its execution, will evaluate the conditions in the table sequentially, one by one. *Customer A*'s spending does not meet the first row's requirement ($\geq$ 1000 EUR), but it does satisfy the second row's spending condition ($\geq$ 500 EUR). Since the loyalty points and account age conditions in the second row are also true, *Customer A* is assigned the "Gold" membership.

2.5. Rule Evaluation in BREs (Inference)

In Business Rule Engines (BREs), inference mechanisms are fundamental to how rules are evaluated and applied. Two well-known methods are **forward chaining** and **backward chaining**. However, many practical BRE implementations, especially those designed for distributed computing environments, employ a **sequential rule execution model** that differs from these classical approaches.

2.5.1. Forward Chaining

Forward chaining is a data-driven approach. It begins with the available facts and iteratively applies rules to derive new facts until no further inferences can be made. In a BRE context, this means that when new data enters the system, the engine scans its set of rules and "fires" those whose conditions are satisfied, thus propagating changes throughout the system. This approach can be highly effective in scenarios where many rules might be triggered by a single change in data, as it allows for the discovery of multiple implications in one pass [67].

To illustrate, an online fraud detection system in a banking context. When a customer performs a transaction, that transaction is treated as an initial fact. The rule engine then iteratively applies a series of rules: for example, if a transaction amount exceeds a predefined threshold, a rule is triggered to mark the transaction as suspicious. This newly derived fact, in turn, generates additional rules such as cross-checking the customer's recent transaction history or validating the transaction's origin. Forward chaining quickly propagates through all relevant rules, aggregating multiple layers of checks and alerts in a single pass.

2.5.2. Backward Chaining

Backward chaining, in contrast, is goal-driven. It starts with a hypothesis or desired conclusion and works backward to determine whether the available data supports that conclusion. In BREs, this method is useful when the system needs to verify whether certain conditions for a decision are met by recursively checking if the underlying data supports the hypothesis. *Wikipedia* explains that backward chaining "begins with a goal and works backwards to see if there are facts that support the goal" [66].

In the context of dynamic pricing, backward chaining can be used to determine whether a product's price should be increased in response to market conditions. The process begins with the goal of "applying a price increase" for products under high demand. The rule engine then systematically works backward to verify the necessary

criteria: it first checks if the current sales data reflect a surge in demand, then verifies that inventory levels are sufficiently low, and finally confirms that competitor prices are trending upward. By focusing solely on the evidence necessary to validate the decision, the engine minimizes unnecessary evaluations and concludes whether the conditions for a price increase are met.

2.5.3. Sequential Approach

While both forward and backward chaining have their benefits, they are not always the best fit for all business rule scenarios. In many practical applications, in particular those involving large datasets in distributed environments, or specific business requirements, the order in which rules are executed is critical. In these cases, a sequential rule execution model is preferred. This model enforces a predetermined order in which rules are applied, ensuring that the output of one rule becomes the input for subsequent ones. Such a deterministic approach is essential when rules have dependencies or when tracking the evolution of data modifications is required for auditing purposes.

For example, IBM's ODM (*Operational Decision Manager*) is a leading BRE that, while often using forward chaining in its core inference engine, also supports the concept of rule ordering to ensure consistent outcomes in complex decision processes [35]. Similarly, many open-source BREs like Drools offer mechanisms for prioritizing rules so that those with critical dependencies execute first. These features highlight that in real-world applications, a strictly sequential execution model is often more practical than classical forward or backward chaining.

For instance, the case of a retail discounting engine that applies multiple, dependent discount rules to a product's price. In this scenario, a retailer may define rules such as a membership discount, a seasonal discount, and a newsletter discount.

Consider the case of a product with an initial price of 1000 EUR. There are three available discounts: a 10% membership discount, a 50 EUR seasonal discount for products over 500 EUR, and a 15% newsletter discount. Using a sequential execution model, the membership discount is applied first, followed by the seasonal discount, and finally the newsletter discount. The newsletter rule, for example, is designed to be mutually exclusive; it is only applied if neither the membership nor the seasonal discount has already modified the price for that row. The execution of the rule engine would proceed as follows, step by step:

- **Step 1:** Applies the membership discount (10% of 1000 EUR = 100 EUR). The price is now 900 EUR.
- **Step 2:** Applies the seasonal discount (50 EUR). The price is now 850 EUR.
- **Step 3:** The newsletter discount is not applied, as the price has already been modified by the previous discounts.

Therefore, the final price of the product after running the engine with the sequantial application of discounts is 850 EUR. Figure 2.1 portrays the logic clearly.

This strict ordering ensures that each rule's effect is correctly layered over the previous ones, thus preserving the intended business logic.

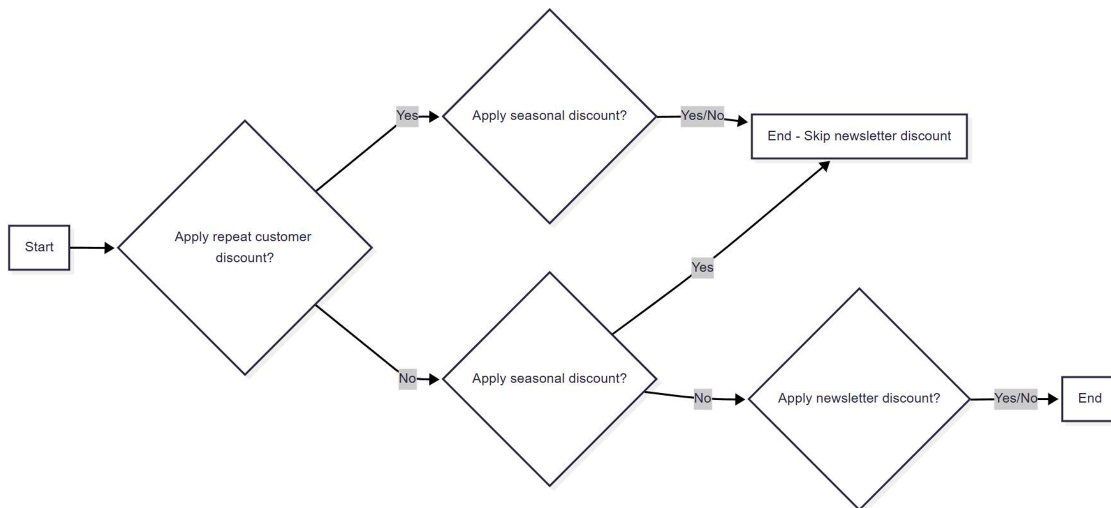

Figure 2.1: Discounts application diagram.

2.6. Hit Policies in BREs

Business Rule Engines often incorporate hit policies to manage how rules are applied when multiple conditions are met for a single data element (or "row" when working with DataFrames). Hit policies determine whether only one rule should fire or if multiple rules can be executed on the same row. For example, some BREs employ a *FIRST HIT* policy, where once a rule has been applied to a row, no other rule is evaluated for that row. This approach simplifies decision-making and prevents conflicting actions by ensuring that only the highest-priority or first-matched rule is executed. In contrast, other systems use a *COLLECT* policy, where every rule that applies to a row is executed, and the results are aggregated to produce a final outcome. Rules can therefore be defined in a way that aligns with specific business needs.

These hit policies are closely linked to standards such as DMN, which explicitly defines hit policies like *UNIQUE*, *FIRST*, *PRIORITY*, and *COLLECT* to guide the execution of decision tables. The DMN standard provides a formal framework that many BREs follow [46].

Choosing a specific hit policy is important because it affects how the rule engine behaves when more than one rule applies to the same input. This is useful because some use cases need a clear, single outcome, while others benefit from combining results from multiple rules.

2.7. BREs in Distributed Computing

Traditional Business Rule Engines (BREs) are predominantly designed for centralized architectures, where all data and processing reside on a single server or a small cluster of tightly coupled machines. This design paradigm works well for many

applications, but it becomes a bottleneck when dealing with the massive datasets and high throughput requirements of modern big data applications.

The need for scalability and parallel processing in big data environments is critical for several reasons. First, the sheer volume of data generated by businesses today often exceeds the capacity of single-server systems [10]. Second, many data-intensive applications require near real-time processing, which can only be achieved through parallel execution [31]. Third, cloud computing environments, which are becoming increasingly popular, are inherently distributed, needing solutions that can utilize distributed resources effectively [2].

Existing BREs often struggle to adapt to these distributed environments. Many are designed with single-node execution in mind, leading to performance reductions and inefficiencies when applied to large-scale datasets.

2.7.1. Cloud Services

AWS offers two main ways to run Spark workloads: **AWS Glue**, a serverless ETL (*Extract, Transform, and Load*) service, and Amazon EMR (*Amazon Elastic MapReduce*), a managed cluster platform. Both can execute the same Spark code, but they differ in how you provision, scale, and pay for resources: factors that directly affect the operational costs, and performance characteristics of the rule engine.

AWS Glue is a fully managed serverless service designed primarily for ETL operations. It abstracts away infrastructure management by automatically provisioning the resources needed to run Spark jobs. To run a rule engine on Glue, the PySpark code can be uploaded (or stored in an S3 bucket) and the job configurations defined through the Glue console. Glue is particularly convenient because it handles scaling and shuts down resources automatically once a job finishes [9]. However, it is less configurable than EMR and may incur cold start delays, which can be noticeable if jobs are run infrequently [8].

On the other hand, Amazon EMR gives users full control over the underlying Spark cluster. It allows configuration of *EC2* instance types, cluster size, networking, and even the installation of custom software. This makes EMR more suitable for advanced use cases that require persistent clusters, fine-tuned performance, or specific Spark configurations [8]. Unlike Glue, EMR clusters can be long-lived, which may reduce latency for frequent executions of a rule engine. The main downside is the increased operational overhead, such as managing uptime, monitoring, and ensuring proper resource utilization.

In terms of pricing, Glue operates on a pay-per-use model based on the number of DPUs (*Data Processing Units*) used per job [9], whereas EMR pricing is based on the EC2 instances running in the cluster along with a small EMR management fee [8]. Depending on the execution frequency and scale of the workload, one option may be more cost-effective than the other.

2.8. BREs and Big Data

2.8.1. The Four V's of Big Data

Big data is commonly characterized by four key dimensions known as the "Four Vs": volume, velocity, variety and veracity [17]. These aspects help define the scale and complexity of modern data environments.

- **Volume** refers to the massive amount of data generated daily, from sources like transactions, mobile devices, and social media. With estimates in the quintillions of bytes created each day, big data systems must be able to scale storage and computing resources accordingly [17].
- **Velocity** captures the speed at which data is produced and needs to be processed. From streaming data in stock markets to real-time sensors in vehicles, fast data flows demand rapid analytics to enable timely decisions like fraud detection or predictive maintenance.
- **Variety** encompasses the many formats and sources of data: structured, semi-structured, and unstructured, ranging from text and video to sensor readings and social media activity. A robust big data solution must handle this diversity smoothly.
- **Veracity** addresses the trustworthiness and quality of data. Inconsistent or inaccurate data can lead to poor decisions, yet organizations often rely on imperfect data due to a lack of better alternatives.

Though additional Vs such as value, visualization, and validity are sometimes discussed, the Four Vs remain the most widely accepted framework for understanding big data challenges, particularly in contexts like Business Rule Engines and analytics platforms [17].

2.8.2. How Inference and Rule Application Differ Between Traditional (Legacy) and Data-Intensive Rule Engines

Traditional or legacy BREs often focus on complex inference mechanisms like forward and backward chaining, optimized for scenarios where the rule set is relatively small and the data volume is manageable. These engines excel at handling intricate logic and decision trees, often used in applications like expert systems and traditional business process management [45].

Data-intensive rule engines, on the other hand, are designed to handle massive datasets and high throughput. These engines prioritize scalability and performance, often employing distributed computing frameworks. Instead of complex inference, they often use simplified rule execution models, such as sequential rule processing or parallel rule evaluation, to achieve high throughput. They are optimized for applications where rules need to be applied efficiently to large volumes of data, such as fraud detection, real-time analytics, and large scale data transformation.

This difference highlights a core challenge: the incompatibility of traditional BRE architectures with the demands of big data. The fundamental architectural mismatch arises because traditional BREs were optimized for managing rule complexity and ensuring logical consistency within a bounded, typically centralized context [45]. In contrast, big data systems prioritize data distribution, parallel processing, and horizontal scalability to handle massive datasets [28]. The very design choices that ensured consistency and maintainability in the traditional model, such as centralized rule repositories and state management, transform into critical liabilities when applied to distributed data. Attempting to apply a centralized engine requires either moving enormous volumes of data to the engine, which violates data locality principles and creates network bottlenecks, or attempting to distribute an architecture not designed for it, leading to complex synchronization problems and state management challenges. [41]

2.8.3. Efficiency in Execution of Business Rules in Distributed Environments

To address the limitations of traditional BREs in big data contexts, there is a need for efficient execution of business rules within distributed computing environments. Distributed frameworks like *Apache Spark*, *Apache Flink*, *Apache Beam* and *Hadoop Map Reduce* have emerged as solutions capable of handling large-scale data processing by enabling parallel execution of tasks across clusters of machines.

2.8.3.1. Apache Spark vs Apache Flink

Apache Flink and Apache Spark are widely used for distributed data processing and support large-scale rule execution.

Apache Flink is designed for high-throughput, low-latency data stream processing. It allows for the parallel execution of tasks by splitting them into multiple parallel instances, each processing a subset of the data. This parallelism is configurable, enabling Flink to efficiently manage large volumes of data by executing tasks concurrently across distributed resources [4] [63].

Similarly, Apache Spark provides capabilities for distributed data processing, supporting parallel execution of operations across a cluster. While Spark is optimized for batch processing, it also offers structured streaming for real-time data processing needs [15] [39]. Both frameworks facilitate the application of business rules at scale, making them suitable for big data environments where traditional BREs may fall short.

Although Apache Flink excels at high-throughput, low-latency stream processing, this project is focused on batch processing rather than real-time data. Spark's strong support for distributed data processing and its well-established integration with cloud environments simplify implementation and maintenance for large-scale workloads. Its in-memory processing and robust scalability make it particularly well-suited for the goal of this project.

2.8.3.2. Apache Spark vs Apache Beam

Apache Beam is a framework that lets users write pipelines in a unified programming model and run them on different backends, like Spark Flink, or Google Cloud Dataflow. This makes it portable and very good for environments where code needs to be deployed to different platforms. Beam is especially known for its support for streaming data, but it also works with batch processing [3]. However, writing Beam pipelines in Python can be more verbose and less natural when compared to PySpark's SQL-style transformations, especially for column-based operations like the ones typically found in a rule engine. As for performance, Beam has more abstraction overhead, and when used with the Spark runner, it tends to be slower than pure PySpark code [60] [58].

Given that this project focuses on batch processing and uses expressions directly applied to data columns, PySpark is a more natural fit. It is easier to write, performs well, and integrates directly with the tools already being used. Beam is still worth considering in cases where future requirements include real-time rule execution or if there's a need to run the same logic across different platforms.

2.8.3.3. Apache Spark vs Apache Hadoop MapReduce

Apache Hadoop MapReduce is a programming model designed for processing large-scale datasets across distributed computing environments. It enables the development of applications that can process vast amounts of data (multi-terabyte datasets) in parallel on large clusters of commodity hardware in a reliable, fault-tolerant manner [5] [65].

The MapReduce model divides a task into two primary functions:

- **Map:** Processes input data and produces a set of intermediate key-value pairs.
- **Reduce:** Aggregates the intermediate data to produce the final output.

This approach allows for efficient, parallel processing of large datasets by distributing tasks across multiple nodes in a cluster.

In the context of Business Rule Engines (BREs), Hadoop MapReduce can be utilized to apply complex rule sets to extensive datasets. However, its batch-oriented nature and higher latency may not be ideal for scenarios requiring real-time or low-latency processing. Spark has been proved to be up to ten times faster than Hadoop under certain circumstances and normal conditions [1].

In many organizations, Hadoop systems have largely remained as passive data lakes, used mainly for storage rather than active analytics. In contrast, Apache Spark has gained traction due to its higher performance, user-friendly APIs, and versatility across different types of analytics workloads [26].

Modern business environments increasingly demand faster, more agile analytics, often in real-time or embedded directly within transactions. As data science becomes more central to operations, there is growing pressure to use analytic resources efficiently while minimizing delays. Legacy systems, often rigid and bound to a platform, struggle to support these needs and lead to fragmented solutions that require manual coordination.

Apache Spark emerges as a solution well-suited to these challenges, offering the flexibility and speed needed to support evolving business strategies [17].

2.8.3.4. Spark's Integration with existing BREs

No commonly used BREs offer the capability of running an engine directly within Apache Spark, nor were they developed primarily with this intention in mind. There are other approaches which explore the idea of combinining the strengths of BREs with the distributed computing power of Spark, but these are simply offered as another implementation possibility. These engines were designed without being specifically built to run on native Spark, therefore not always taking fully advantage of Spark's capabilities. Among these, two prominent Business Rules Engines, IBM Operational Decision Manager (ODM) and Drools, have received attention for their potential integration with Spark.

IBM's ODM, a commercial BRMS offering, in their official documentation [32], has previously provided examples of how decision services could be invoked within Spark jobs, using Spark RDD (*Resilient Distributed Dataset*) transformations, as can be seen in Figure 2.2.

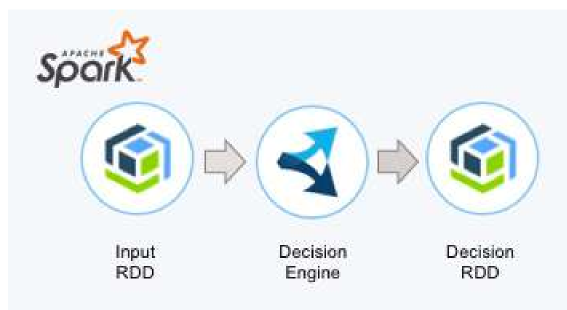

Figure 2.2: IBM ODM Spark integration.
From decisions-on-spark, on GitHub [49].

This is primarily done through reading decision requests into RDDs from various data sources, applying a transformation function to the RDDs that invokes the ODM decision service (either through RESTful APIs or embedded sessions), generating a new RDD containing the outcomes of the decision service for each input record, and storing the resulting RDD in a persistent storage system such as HDFS (*Hadoop Distributed File System*) [32], ensuring high scalability by the cluster deployment [53]. Apache Spark compiles RDD transformations and actions into Java bytecode, meanwhile IBM ODM compiles business rules into Java bytecode thanks to its Decision Engine. However, it is important to note that this integration now appears to be deprecated. IBM's original documentation describing the architecture and implementation of Spark-ODM integration [32] is no longer available through official IBM channels and can only be accessed through third-party archives (such as `archive.org`). An example implementation of this integration is still available in the *decisions-on-spark* project on GitHub [49], which demonstrates loan application validations using ODM within a Spark environment.

This lack of current support and visibility suggests that IBM may no longer maintain or recommend this approach.

In contrast, Drools, a widely-used open-source BRE maintained under the *KIE* community, remains an actively developed and viable option for Spark integration. Drools supports integration with both batch and streaming components of Spark through flexible Java APIs. However, Spark is not even mentioned in their documentation. Although this may not be necessarily negative, it may also pose limitations to the full usage of Spark's capabilities, in a rather similar way to IBM's offer.

Despite the lack of direct support for Spark, Drools can indeed be integrated with Spark through the use of Java APIs. There are many examples of this integration available on GitHub, such as the *spark-drools-example* project [16], which demonstrates how to use Drools rules within a Spark application.

The project is a barebones example of how to use Drools rules within a Spark application. It demonstrates how to set up a Spark application, and the peculiarities of using Drools rules within this context. For example, it shows that to avoid the re-loading and re-compilation of rules, the rules should be first broadcasted to all workers in their compiled form. As for the rest of the code, it is similar to the standard Drools code, with the exception of the fact that the rules are executed in parallel across the Spark cluster.

2.8.3.5. PySpark

PySpark is the Python API designed for Apache Spark. It enables developers to write Spark applications using the Python language, granting access to the extensive features and distributed data processing capabilities of the Spark framework [6]. Notably, the core Apache Spark engine itself is primarily written in *Scala* and executes on the Java Virtual Machine (JVM). PySpark acts as an interface layer, allowing Python developers to leverage Spark's computational power while working within their familiar programming environment and utilizing standard Python libraries [18]. This integration often facilitates easier or more rapid development compared to using Scala, given Python's widely regarded accessible syntax and gentler learning curve.

No current BREs, even those that offer indirect implementation with Spark, support PySpark.

2.8.4. Limitations of Current Open-Source Rule Engines in Big Data Processing

Despite the advancements in distributed processing frameworks, current open-source rule engines are not designed to be applied to big data processing. Many traditional rule engines are not inherently designed for distributed execution, leading to integration challenges with platforms like Spark.

As mentioned in the previous subsection, only IBM's ODM and Drools offer the possibility to integrate Spark, although IBM's offering may have already deprecated its Spark integration. On top of that, IBM's licensing cost is on the high end of the spectrum, considering its dedication to commercial use, while being strictly closed-

source. Afterall, only Drools seems to remain a viable alternative.

Even then, Drools' integration with Spark is not straightforward, as it requires a certain level of knowledge of the inner workings of both Drools itself, and Spark's implementation through Java. This can lead to a steep learning curve for the average user of a BRE, who often lacks the technical expertise, and is dedicated to business logic rather than the underlying technology. Moreover, this integration is not officialy supported by Drools but is instead derived from a community contribution.

2.9. Existing Business Rule Engine Solutions

Several Business Rule Engines (BREs) have been developed to cater to different industries, offering various approaches to rule management, inference mechanisms, and system integrations. Some of the most widely used BREs include *IBM Operational Decision Manager (ODM)*, *Drools*, and *FICO Blaze Advisor*.

IBM ODM is a leading commercial rule engine that provides extensive rule management features, including version control, auditing, and real-time execution capabilities. Designed for enterprise-scale applications, it integrates with IBM's cloud and on-premise solutions. Its primary inference mechanism is forward chaining, where rules are applied sequentially based on available data to drive automated decisions [35].

Drools, on the other hand, is a widely used open-source BRE that supports both forward and backward chaining. This flexibility allows businesses to implement different rule execution strategies depending on their needs. Drools is particularly favored in Java-based environments due to its seamless integration with Java applications, offering developers a powerful framework to define and manage rules programmatically [54]. Unlike IBM ODM, which primarily targets large enterprises, Drools is often used by organizations that require an open-source and customizable solution for rule-based automation.

FICO Blaze Advisor is another prominent commercial rule engine, widely used in financial services due to its advanced capabilities in risk assessment and compliance enforcement. It enables businesses to define complex rule sets while maintaining high performance and scalability. Blaze Advisor primarily operates using forward chaining, allowing it to evaluate multiple rules dynamically based on incoming data. It is particularly useful in automating credit scoring, loan approvals, and fraud detection, where rapid decision-making is critical [22].

There are many other alternatives worth mentioning, designed to attend to different business needs. One such example is *DecisionRules.io*, a cloud-based BRE designed for ease of use, offering a no-code interface alongside API-based integration for developers. It is particularly suited for businesses looking to implement rule-based decisioning with minimal technical overhead, making it a competitive alternative to enterprise-heavy solutions like IBM ODM and FICO Blaze Advisor [19].

Another option is *Pega Decision Management*, which combines business rules with AI models to help companies make smarter decisions. It is useful in areas such as customer service and marketing, where decisions can benefit from both predefined

rules and real-time predictions [48]. *OpenL Tablets* is also worth mentioning, since it is an open-source tool that lets users write rules in a spreadsheet format, making it easier for non-developers to manage logic without diving into code [47].

2.9.1. Table Summary

A comparative overview of the Business Rule Engines (BREs) discussed previously is presented in tabular format in Tables 2.2 and 2.3, showing key characteristics such as BRE type, scalability, rule representation (with proprietary renamings often used by the vendors), inference mechanism, integration capabilities, typical use cases, distributed computing support, ease of use, and target audience.

An extra column has been added mentioning how *Stickler* would compare to the other BREs. At this point, it is important to note that the entries in this column represent a conceptual framework for *Stickler's* functionality, based on the project's objectives.

Charac-teristic	**IBM Operational Decision Manager (ODM)**	**Drools**	**FICO Blaze Advisor**
Type	Commercial	Open-source	Commercial
Scalability	High, enterprise-scale	Moderate, Java-based	High, financial services focused
Rule Represen-tation	If-then rules (*BAL*) and decision tables	If-then rules (*DRL*) and decision tables (spreadsheet that generates DRL)	If-then rules (*SRL*) and decision tables
Inference Mechanism	Forward chaining or sequential	Forward and backward chaining	Forward chaining
Integration	IBM cloud and on-premise solutions, API	Java applications, API	Financial services systems, API
Use Cases	Enterprise rule management, auditing	Customizable rule automation	Risk assessment, compliance
Distributed Computing Support	Limited without IBM specific extensions	Limited, requires custom implementation	Limited without specific integrations
Ease of Use	Complex, enterprise-level	Moderate, developer-focused	Complex, financial domain specific
Target Audience	Large enterprises. Government	Organizations that need a customizable solution. Open-source community	Financial institutions

Table 2.2: BREs comparison table.

Characteristic	DecisionRules.io	Stickler
Type	Cloud-based	Open-source
Scalability	Moderate, cloud-native	High
Rule Representation	If-then rules and decision tables (both with visual editor)	If-then rules and decision tables (equivalent)
Inference Mechanism	Primarily forward	Sequential execution
Integration	API-based, no-code interface	Python integration
Use Cases	Easy-to-use rule-based decisioning	Decisions over large datasets
Distributed Computing Support	Cloud native, better support than others listed	Full support, on-premise or cloud based
Ease of Use	User-friendly (good GUI), no-code options	Moderate (no GUI)
Target Audience	Businesses and start-ups, seeking quick rule implementation	Enterprises with high volumes of data

Table 2.3: BREs comparison table, continuation.

Chapter 3

System Design and Architecture

3.1. Problem Definition

As outlined in Chapter 2, traditional Business Rule Engines (BREs) often rely on inference mechanisms such as forward and backward chaining. While these approaches are well-suited for reasoning over small to medium-sized datasets, they struggle when applied to large-scale distributed data processing. Most conventional BREs are not designed to handle the computational demands of big data frameworks like Apache Spark, which operate on massive datasets across distributed environments.

One key limitation of existing solutions is that they tend to be optimized for single-node execution or small-scale enterprise workflows. These systems often assume that all data is available in memory, which is impractical when processing terabytes or petabytes of data. In contrast, PySpark enables distributed computing across clusters, but existing rule engines do not integrate well with its RDD (Resilient Distributed Dataset) model or DataFrame-based execution, limiting their effectiveness in high-volume, cloud-based workloads.

Another major challenge is efficient rule execution in a distributed setting. Rule engines on the market rely on sequential processing, where rules are executed in a predefined order, often requiring iterative passes over the data. This becomes computationally expensive in big data scenarios. Furthermore, handling dependencies and cascading rule effects in a distributed environment introduces additional complexity. Ensuring that rules are applied in the correct sequence while maintaining performance is a non-trivial challenge.

Given these issues, there is a clear need for a scalable, data-intensive BRE that can efficiently apply business rules to large datasets in a distributed computing environment. Such a system must be designed to leverage parallel execution, minimize redundant computations and integrate well with Spark workflows. The reason behind the decision of using PySpark and not Spark on Java (or any other programming language, such as Scala), is due to Python's widespread use on big data, and for its simpler syntax.

The proposed rule engine in the next sections aims to address these gaps by optimizing rule evaluation strategies, ensuring scalability, and enabling cloud-based

deployments.

3.2. Requirements and Design Considerations

3.2.1. Functional Requirements

The rule engine must provide a robust yet user-friendly way to define and execute rules over large datasets represented as PySpark DataFrames. To achieve this, it should support:

- **JSON-based rule definition**: Rules should be easily configurable in a structured, human-readable JSON (*JavaScript Object Notation*) format, allowing users to define and modify rules without deep programming nor advanced data engineering knowledge. This ensures that rule management is not restricted to senior engineers alone.
- **Validation and error handling**: The system must verify that rules are correctly defined before execution. This includes:
  - Checking for syntax errors in the JSON definition. Such as:
    - The proper fields are present.
    - The correct data types are used.
    - The JSON structure is valid.
  - Providing clear and descriptive error messages when a rule is misconfigured or references an invalid column.
  - Ensuring that names do not collide and preventing reserved names from being used.

  We can distinguish between three types of validation:

  - **Name validation**: If a rule references a column that does not exist either in the input DataFrame or in the DataFrame that is being computed, an error should be raised. Also if the name matches another column name, or is empty. For example:
    - "Error on `{rule_name}` definition: the rule name cannot be empty", for an empty rule name.
    - "Error on `{rule_name}` definition: the rule name is already in use. Please choose a different name", for a rule name that is already in use.
    - "Error on `{rule_name}` definition: the rule name is the same as a column in the history DataFrame. Please choose a different name", for a rule name that is the same as a column in the history DataFrame.
  - **Expression validation**: If an action or condition contains a Spark SQL invalid expression, such as "price plus 10" instead of "price + 10", an error should be raised. For example: "Error on `{rule_name}` definition: the

expression {“price plus 10”} is invalid. Please use a valid operator and syntax”.

- **Reference validation**: If any column referenced in a rule’s conditions or actions does not exist in the input DataFrame, nor has been created by a previous rule, an error should be raised. For example: “Error on `{rule_name}` definition: the column `price` doesn’t exist”.

- **Efficient execution on distributed data**: Rules must be applied at scale using PySpark, ensuring optimal performance without excessive overhead.

- **Dependency management**: If a rule relies on the results of a previously executed rule, the engine must ensure correct execution order and prevent race conditions.

- **Transparent rule application history**: The engine must maintain a detailed history of rule applications. This is important because:
  - It enables **traceability**, allowing users to audit which rules were applied to each data row.
  - It supports **debugging** and troubleshooting by making it clear how data was transformed at each step.
  - It allows rollback or review of intermediate results, which is valuable for iterative development and **validation**.

- **Cloud integration**: The system should be cloud-friendly, allowing integration with services such as AWS, Google Cloud, or Azure for seamless deployment in distributed computing environments. For this, it may be necessary to package the application, to then be installed with a package manager such as *pip*.

3.2.2. Non-Functional Requirements

Beyond core functionality, the engine must also meet performance, scalability and reliability criteria:

- **Scalability**: The engine must handle massive datasets efficiently, leveraging Spark’s distributed nature. It should support parallel execution while ensuring consistency in rule application.

- **Performance optimization**: The rule evaluation process should be optimized to minimize redundant computations and avoid unnecessary recomputation of intermediate results.

- **Deterministic execution**: Given that rule execution order matters, the system must guarantee that the same input data and rules always yield the same results, preventing unexpected variations in output.

3.2.3. Trade-offs: Inference Complexity vs. Large-Scale Execution

Traditional rule engines often focus on complex inference mechanisms such as forward or backward chaining, which work well for small rule sets but can become inefficient in big data scenarios. This engine, however, prioritizes efficient execution over large-scale datasets rather than deep inference, meaning it will implement **sequential execution**. Key trade-offs include:

- Optimizing for speed and scalability rather than extensive logical inference.
- Sequential or parallel execution models instead of complex reasoning-based evaluation.
- Explicit dependency resolution rather than implicit rule inference to ensure clarity and efficiency.

3.3. Architecture Overview

3.3.1. Design Constraints in Rule Logic

In this engine, rules are formulated as straightforward **if-then** statements. This approach ensures that each rule clearly specifies its conditions and the corresponding actions to be executed once those conditions are met. Moreover, rule execution is carried out sequentially, mirroring the structure of decision tables. In practice, this means that the outcome of one rule is definitively established before the next rule is evaluated. Such a sequential and table-like definition not only simplifies both the creation and verification of rules but also facilitates the clear tracking of dependencies and the evolution of data transformations within the system.

Actions and Conditions

Every rule is composed of two fundamental components: a set of **conditions** and a set of **actions**.

The engine processes rules against each data row as follows:

- If all of a rule's conditions evaluate to true for a given row, its specified actions are executed on that row.
- Optionally, an **"otherwise"** clause can be defined for a rule. If the primary conditions are not met and an "otherwise" clause exists, the actions within this clause are executed.
- If a rule's conditions are not met and no "otherwise" clause is provided, the row remains unchanged by that particular rule.

This conditional logic provides flexibility in defining complex behaviors.

The capabilities of actions within this engine are constrained to ensure predictability:

- Actions are strictly limited to **creating new columns or modifying existing columns** within the dataset.

- By design, actions **cannot directly trigger other rules** within the engine, invoke external services (like APIs), or send notifications. This limitation is intentional, as it ensures that the rule engine remains self-contained and portable. These types of integrations must be implemented by the user in a post-processing step, after the rule engine has completed its execution, using the output DataFrame generated.

3.3.2. Rule Execution Types

In this implementation, the concept previously known as "hit policy" has been refined and is now referred to as "rule execution type". This determines how rules are processed and their outcomes applied. Each rule is configured to operate in one of two primary modes: **accumulative** or **cascade**. The **cascade** mode, in particular, is implemented with a flexible mechanism using **cascading groups** to manage rule interactions.

Accumulative Mode

A rule operating in **accumulative** mode is applied to every data row that satisfies its conditions. The rule's actions are executed for each matching row, and any modifications are reflected in the dataset, potentially influencing subsequent rule evaluations or forming part of the final output. This mode is comparable to a "collect" policy, where multiple rules can independently contribute their changes to a row.

Cascade Mode

In contrast, **cascade** mode introduces a more controlled execution flow using **cascading groups**. A cascading group is essentially a named set of rules. If a rule is part of a cascading group and is set to "cascade", it will execute on a given row *only if no other rule within that same cascading group has already been successfully applied to that row.* This design is crucial for scenarios where only one specific modification should occur from a list of possible alternatives for a particular aspect of the data. It helps prevent conflicting updates and ensures that rule execution remains deterministic and clear, as typically the first rule (often based on evaluation order) in the group that triggers will take effect.

An example of this execution type, would be an online store applying different discounts.

- A "Repeat Customer Discount" rule might belong to a cascading group named "CustomerTierDiscounts".

- A "Seasonal Promotion Discount" rule could be in a different cascading group, "EventDiscounts".

Since these rules are in separate cascading groups, a customer qualifying for both could receive both discounts, as the groups operate independently for this purpose.

Now, there is a "Newsletter Signup Discount" that should be mutually exclusive with *any* of the aforementioned discounts. To achieve this, the "Newsletter Signup Discount" rule would be configured as a member of *both* the "CustomerTierDiscounts" group *and* the "EventDiscounts" group.

- If the "Repeat Customer Discount" (from "CustomerTierDiscounts") has already been applied to a customer's record, the "Newsletter Signup Discount" will not be applied, because another rule from one of its designated groups ("CustomerTierDiscounts") has already executed for that record.
- Similarly, if the "Seasonal Promotion Discount" (from "EventDiscounts") has been applied, the "Newsletter Signup Discount" will also be skipped.

3.3.3. Input and Output

Input

There are three main inputs to the rule engine:

- A **DataFrame** containing the data to be processed.
- A **JSON file** defining the rules to be applied.
- A dictionary of **user-defined functions** for the expressions evaluating conditions and actions. This is optional, only needed if the user wants to use custom functions in the rules.

The rule engine ingests a PySpark dataframe, which is the input data to be processed. A dictionary of UDFs (*User-Defined Functions*) can also be provided, which can be used to evaluate the conditions and actions of the rules. This dictionary should contain the names of the UDFs as keys and the UDFs themselves as values.

User-Defined Functions (UDFs) in PySpark provide a way to extend Spark's functionality [6]. They allow implementing custom logic using Python when Spark's native functions or DataFrame API don't cover a specific transformation, such as custom parsing or a specialized calculation. A UDF can be called directly within DataFrame operations like `select`, `withColumn`, or `expr`. Internally, Spark serializes the data from each row, passes it to the Python function, and receives the result back into the distributed process.

The key advantage of UDFs is their flexibility, enabling the implementation of almost any required transformation. They are particularly relevant for domain-specific tasks. However, they can introduce performance overhead due to the serialization between the Python process and the JVM for each function call. For this reason, they should be avoided unless necessary, as they can significantly slow down the execution of Spark jobs.

A short example of a UDF is shown below, where a simple function is defined to append the string "_udf" to the input value. This function is then registered as a

UDF in PySpark, allowing it to be used in DataFrame. However, it is worth noting that this UDF is not particularly useful, as it is just a simple string concatenation.

Another point to consider is that the engine will take a dictionary of UDFs, registering the UDFs itself: thus the last line of the code should not be executed.

Listing 3.1: UDF Example

```
def my_udf(value):
    return value + "_udf"

my_udf_spark = udf(my_udf, StringType())

spark.udf.register("my_udf", my_udf_spark)
```

Because of UDF's versatility, they can also implement conditions inside the execution of an action. While this is technically feasible, it is generally discouraged for reasons of readability and maintainability.

The rule definition input is provided through a JSON file, which defines all rules along with their respective conditions and actions. The structure of the input follows the schema below:

Listing 3.2: JSON Schema for Rule Input

```
{
    "rules": [
        {
            "rule_name": "rule_name1",
            "execution_type": "cascade"/"accumulative",
            "cascade_group": [0, 1, 2] / 0 (optional),
            "conditions": [
                {
                    "expression": "spark_sql_expression"
                }
            ],
            "actions": [
                {
                    "output_col_name": "column_name",
                    "operation": "spark_sql_expression",
                    "otherwise": "spark_sql_expression" (optional)
                }
            ]
        },
        ...
    ]
}
```

The JSON input is structured as a single object containing a primary key, `"rules"`, which holds an array of rule definition objects. Each rule object within this array defines an individual rule to be processed by the engine. Below is a detailed explanation of each field within a rule object:

- `rule_name` (string): This field provides a unique and descriptive name for the rule (e.g., `"HighValueCustomerDiscount"`). This name is useful for iden-

tification, logging, and debugging of the application of specific rules during execution.

- `execution_type` (string): This mandatory field determines how the rule interacts with other rules when applied to the data. It accepts one of two string values:
  - `"cascade"`: This mode utilizes cascading groups to control rule application. A cascade rule will only execute on a row if no other rule within the same `cascade_group` has already been applied to that row. This is used to enforce precedence or mutual exclusivity.
  - `"accumulative"`: In this mode, the rule is applied to every row that satisfies its conditions. Its actions are executed independently of other accumulative rules, and changes are reflected in the dataset.
- `cascade_group` (integer or array of integers, *optional*): This field is only relevant when the `execution_type` is set to `"cascade"`. It assigns the rule to one or more cascading groups, which are identified by integer values. It defaults to `0` if not specified.
  - If the rule belongs to a single cascading group, this field will contain a single integer (e.g., `0`, `1`).
  - If the rule needs to be a member of multiple cascading groups (for more complex mutual exclusivity scenarios, as discussed in the 3.3.2 section), this field will be an array of integers (e.g., `[0, 1, 2]`).

  For rules with `execution_type` set to `"accumulative"`, this field may be omitted or ignored by the engine. Associating an `accumulative` rule with a cascading group means that execution has a dual effect: its own actions are applied, and it also preempts (blocks) any cascade type rules belonging to the same `cascade_group`(s) that are evaluated later for that data row.
- `conditions` (array of objects, *optional*): This field contains an array of condition objects. For a rule's actions to be triggered for a given data row, *all* conditions specified in this array must evaluate to true (logical AND). If left undefined, or if the array is empty, the condition will default to true, meaning the rule's actions will always be executed for every row. Each condition object has the following structure:
  - `expression` (string): This holds a Spark SQL boolean expression (e.g., `"original_price > 100"`). that will be evaluated against each row of the input data. The expression should resolve to true or false. UDFs can be used within this expression, allowing for custom logic to be applied.
- `actions` (array of objects): This field defines an array of action objects that specify what operations to perform if all the rule's conditions are met. As per the design constraints, actions are limited to creating new columns or modifying existing ones. Each action object has the following structure:

- `output_col_name` (string): This specifies the name of the column that will be created or updated by the action (e.g., `"final_price"`).
- `operation` (string): This contains a Spark SQL expression. If all the rule's main `conditions` are met, this expression is evaluated, and its result is assigned to the `output_col_name` (e.g., `"original_price * 0.90"`). UDFs can be used within this expression.
- `otherwise` (string, *optional*): This optional field also contains a Spark SQL expression. It defines an alternative operation if the rule's main `conditions` are *not* met, and the result is assigned to the `output_col_name`.

  If the rule's main `conditions` are met, this `otherwise` field (if present) is ignored for this action item. UDFs can be used within this expression.

Output

The *Engine* sequentially executes each rule while recording the changes made to the dataset. The engine produces two outputs: a **final, transformed DataFrame** and a **history DataFrame** that transparently logs how each rule modified the data. This dual output allows users to audit the rule execution process, verifying both the final outcome and the intermediate steps taken along the way.

To illustrate how output and history DataFrames are generated, consider a use case involving a product catalog. The initial input DataFrame, as shown in Table 3.1, contains three columns: product name, available stock and unit price.

A rule named `discountPrices` is defined with the purpose of applying a 10% discount to any product priced above 100. This rule modifies the `price` column directly (works both as an input column and an output column).

Listing 3.3: JSON Input Example

```
1   {
2       "rules": [
3           {
4               "rule_name": "discountPrices",
5               "conditions": [
6                   {
7                       "expression": "price > 100"
8                   }
9               ],
10              "actions": [
11                  {
12                      "output_col_name": "price",
13                      "operation": "price * 0.9"
14                  }
15              ]
16          }
17      ]
18  }
```

Product	**Stock**	**Price**
Product A	50	90.00
Product B	20	150.00
Product C	10	120.00

Table 3.1: Original input DataFrame.

After executing the rule engine, the resulting output DataFrame, reflects the discounted prices for qualifying rows, with the original values overwritten. In this case, “Product B” and “Product C” receive the discount, while “Product A”, priced below the threshold, remains unchanged (see Table 3.2).

Product	**Stock**	**Price**
Product A	50	90.00
Product B	20	135.00
Product C	10	108.00

Table 3.2: Output DataFrame after applying `discountPrices` rule.

In parallel, a history DataFrame (see Table 3.3) is generated to track all transformations applied during rule execution. This auxiliary structure includes the original input columns along with two additional columns named `discountPrices` and `discountPrices_Price`, which record if the rule was applied on each row and the price values produced by the `discountPrices` rule, respectively. This design allows for traceability and auditability of rule-based transformations without altering the raw input.

To clarify, each rule generates its own history column, named `rule_name`, while generating an extra column for each of their actions.

Product	**Stock**	**Price**	**discountPrices**	**discountPrices_Price**
Product A	50	90.00	False	90.00
Product B	20	135.00	True	135.00
Product C	10	108.00	True	108.00

Table 3.3: History DataFrame capturing original values and rule-generated modifications.

3.3.4. Rule Execution Process

The architecture of the rule engine should center around a core *Engine class* that orchestrates the entire process. Initially, the *Engine* first registers the UDFs, to then validate the input rules defined in a JSON file, to catch syntax errors and missing dependencies before execution. This early validation is crucial to prevent unnecessary runtime crashes and ensure that only correctly defined rules are processed.

The engine then proceeds to execute the rules sequentially, applying each rule to the input DataFrame. The execution process involves evaluating the conditions of each rule against the data rows. If a row meets the conditions, the corresponding

actions are executed, modifying the data as specified. The engine maintains a history of all transformations, allowing users to trace back the changes made to each row. This history is captured in a separate DataFrame, which includes the original input columns and the new columns generated by the rules. This design ensures that users can audit the rule execution process and understand how the data has been transformed at each step.

This idea can be visualized in the following Figure 3.1.

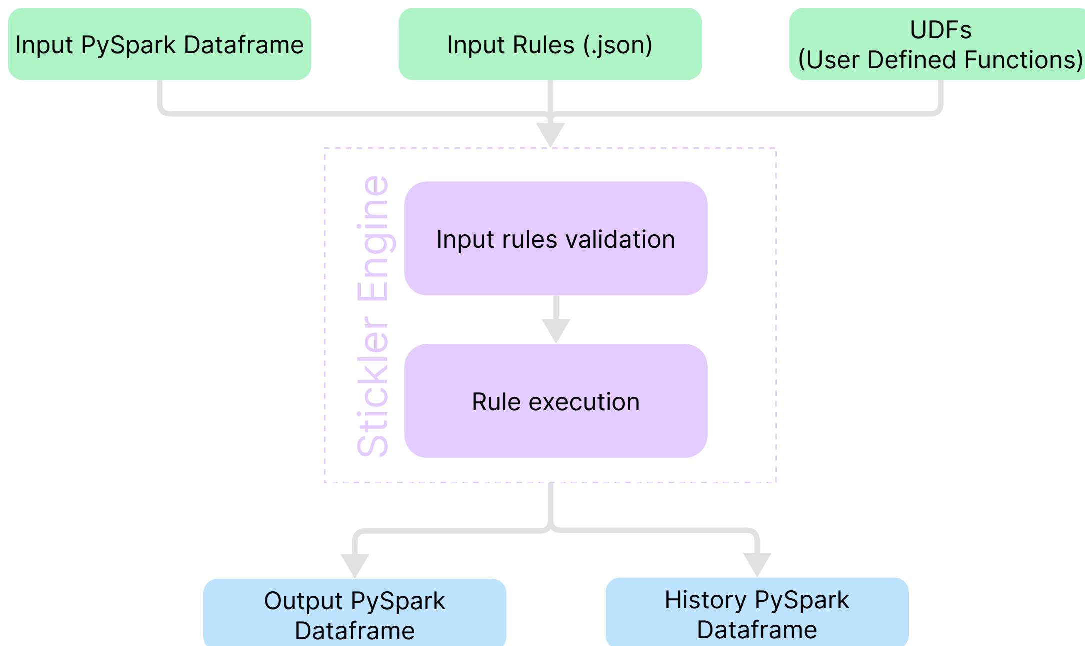

Figure 3.1: Simplified rule engine execution flow.

Chapter 4

Implementation

Following the establishment of the theoretical foundation and system design, this chapter concentrates on the practical implementation of the business rule engine. The implementation phase involves translating the defined requirements into a functional system, ensuring the rule engine's capacity to efficiently process large-scale datasets within a distributed environment.

The code for the rule engine is available on GitHub[1], where an user guide is also provided in the repository's README. This user guide can also be found in Appendix C. The project is licensed under the **GNU General Public License v3.0 (GPLv3)**. This license allows for free use, modification, and distribution of the code, provided that any derivative works are also distributed under the same license and that the original copyright notice and license are included in any copies or substantial portions of the software.

4.1. Technology Stack and Tooling

A key decision in this project is the choice of Spark as the core framework for processing distributed data. Among many reasons, Spark stands out because it is specifically designed to handle large-scale data across clusters with minimal overhead. Its in-memory computing capabilities allow for faster processing, while its inherent support for immutable data structures and built-in partitioning strategies helps eliminate race conditions and ensures fault tolerance [25]. This is highlighted by its widespread adoption: Apache Spark currently is being utilized by more than 80% of Fortune 500 companies for big data processing [25]. These characteristics make Spark an ideal candidate for building a scalable business rule engine that can efficiently process massive datasets in a distributed environment.

In addition to Spark, the technology stack includes several key libraries and cloud services that further enhance the system's capabilities. Specifically, **PySpark**'s high-level API is utilized for efficient data querying and transformation, simplifying complex operations on structured data. The engine also benefits from Python's extensive ecosystem, which includes libraries for auxiliary data manipulation and

[1] `https://github.com/ginacassin/stickler`

testing, as well as the language's ease of use.

To help to parse the JSON rule definitions, the engine utilizes **Pydantic**, a Python library that provides robust data validation and settings management using Python type annotations. Pydantic is employed to enforce a strict schema, ensuring that each rule is assigned a unique name and that its conditions and actions are clearly defined [50].

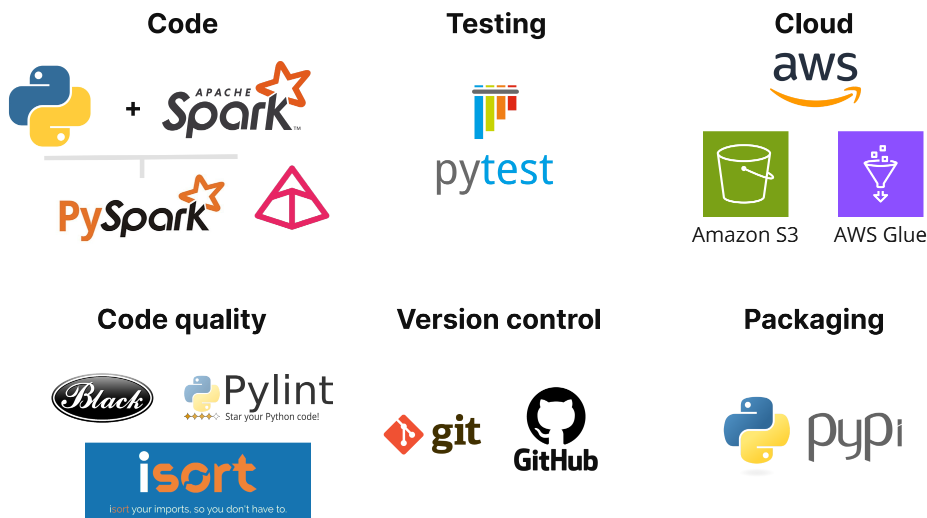

Figure 4.1: Technology stack and frameworks.

Packaging

To ensure accessibility and ease of integration into other Python projects, this rule engine is packaged as a standard Python library. It has been published on PyPI (*Python Package Index*), the official third-party software repository for Python, making it readily available to the wider developer community[2]. This allows engineers to easily incorporate the engine into their own data processing workflows. Installation is managed straightforwardly using *pip*, Python's standard package installer. Users can install the library by executing the following command in their terminal:

Listing 4.1: Installing Stickler via pip

```
1 pip install stickler
```

Once installed, the engine's functionalities can be accessed within any Python script or application through a standard import statement:

Listing 4.2: Importing Stickler in Python

```
1 import stickler
```

[2] https://pypi.org/project/stickler/

Cloud deployment

On the cloud front, to test the engine, it will be deployed on **AWS**, utilizing **AWS S3** for storage and **AWS Glue** for running the PySpark job. Integration with cloud services like this one enables easy access to distributed storage systems and other cloud-native tools, ensuring that the engine remains flexible and cost-effective in a production environment.

It is important to note that while AWS S3 and Glue serve as the environment for testing the engine's cloud deployment, its design as a pip-installable Python library fundamentally offers engineers the versatility to integrate and deploy *Stickler* across a diverse array of other cloud services, such as Google Cloud Platform, Microsoft Azure, or even on-premise infrastructures.

Code style guide and Quality tools

To promote code clarity, maintainability, and consistency, this project conforms to **Google's Python Style Guide**, which outlines best practices for writing clear, maintainable and consistent Python code [27]. By following this guide, the goal is to ensure that the codebase remains clean and readable, especially as it grows or is shared with others. For example, the guide recommends using lowercase letters with underscores for function and variable names (e.g., *apply_rules()*), and using CapWords for class names (e.g., *RuleEngine*). It also promotes writing short, focused functions and including docstrings in a specific format to explain clearly their purpose and usage.

To further enhance code quality and automatically enforce styling conventions, this project integrates several industry-standard Python linting and formatting tools: **isort**, **black** and **pylint**.

isort is a Python utility focused on organizing imports within Python files. It achieves this by sorting them alphabetically and automatically separating them into distinct sections based on their type. The tool offers extensive configuration options, allowing it to be adapted to virtually any specific style guide requirements [64]. This systematic arrangement of imports is particularly beneficial in larger projects for managing dependencies and enhancing the overall readability of the code.

black serves as a Python code formatter. It operates on the principle of enforcing a single, consistent code style with very few configuration options. This approach is intentionally designed to eliminate discussions and debates related to code formatting, thereby allowing developers to concentrate on more critical aspects of software development [13]. *black* automatically reformats files according to its strict style, which is a superset of PEP 8, ensuring uniformity across the entire codebase.

pylint is a versatile static analysis tool for Python. It performs a wide range of checks, including identifying programming errors, ensuring adherence to coding standards, detecting code smells (indicators of deeper problems in the code), and offering suggestions for potential refactoring. Furthermore, *pylint* can provide insights into the complexity of the code [51].

To simplify the execution of these code quality tools, a **Makefile** has been created. This allows for the combined invocation of *isort*, *black*, and *pylint* through a

single command: `make code-check`. The content of the file is as follows:

Listing 4.3: Makefile for Code Quality Checks

```
.PHONY: code-check

code-check:
	isort .
	black .
	pylint src tests
```

Testing

For testing, the project employs **pytest**, a powerful and flexible testing framework for Python. *pytest* simplifies the process of writing and executing tests, providing a rich set of features such as fixtures, assertion introspection and plugins. It is particularly well-suited for unit testing, integration testing, and functional testing, making it a popular choice among developers [52].

Version control

The project is managed using **Git**, a distributed version control system that allows for efficient tracking of changes in the codebase. Git enables multiple developers to collaborate on the same project, facilitating branching, merging and versioning of the code. The project is hosted on **GitHub**, a web-based platform that provides Git repository hosting, along with additional features such as issue tracking, pull requests, and project management tools.

4.2. Architecture

The architecture of the rule engine system is designed for flexibility and scalability, leveraging PySpark for distributed data processing. While the core engine is a portable Python library, its capabilities will be tested within an AWS cloud environment, as it will be further explained in Chapter 5. Nonetheless, the engine can be deployed in other environments (such as Google Cloud Platform, Microsoft Azure or even on-premise), as long as these environments support PySpark and a similar implementation to the proposed one is followed.

The operational flow begins with the necessary **inputs**:

- An **Input PySpark DataFrame**: This is the primary dataset to which the rules will be applied. While it must be in PySpark DataFrame format for the engine, the original data can be sourced from various storage systems, such as files in Amazon S3, which can be in CSV (*Comma Separated Values*) or Parquet format (Apache's columnar storage file format). This is left to the user to decide, as the engine is agnostic to the data source.
- An **Input Rules Configuration File**: This is a JSON file (`rules.json`) that defines all the business rules, including their names, conditions, actions,

execution types, and any cascade logic. This must be compliant to the schema defined in the engine, as outlined in Section 3.3.3.

- An optional **UDF Python File** (`udf.py`): If the rules require custom logic not available through standard Spark functions, users can provide a Python file containing their User Defined Functions (UDFs). These UDFs are registered with Spark by the engine and can be referenced within rule expressions.

The proposed architecture is illustrated in Figure 4.2.

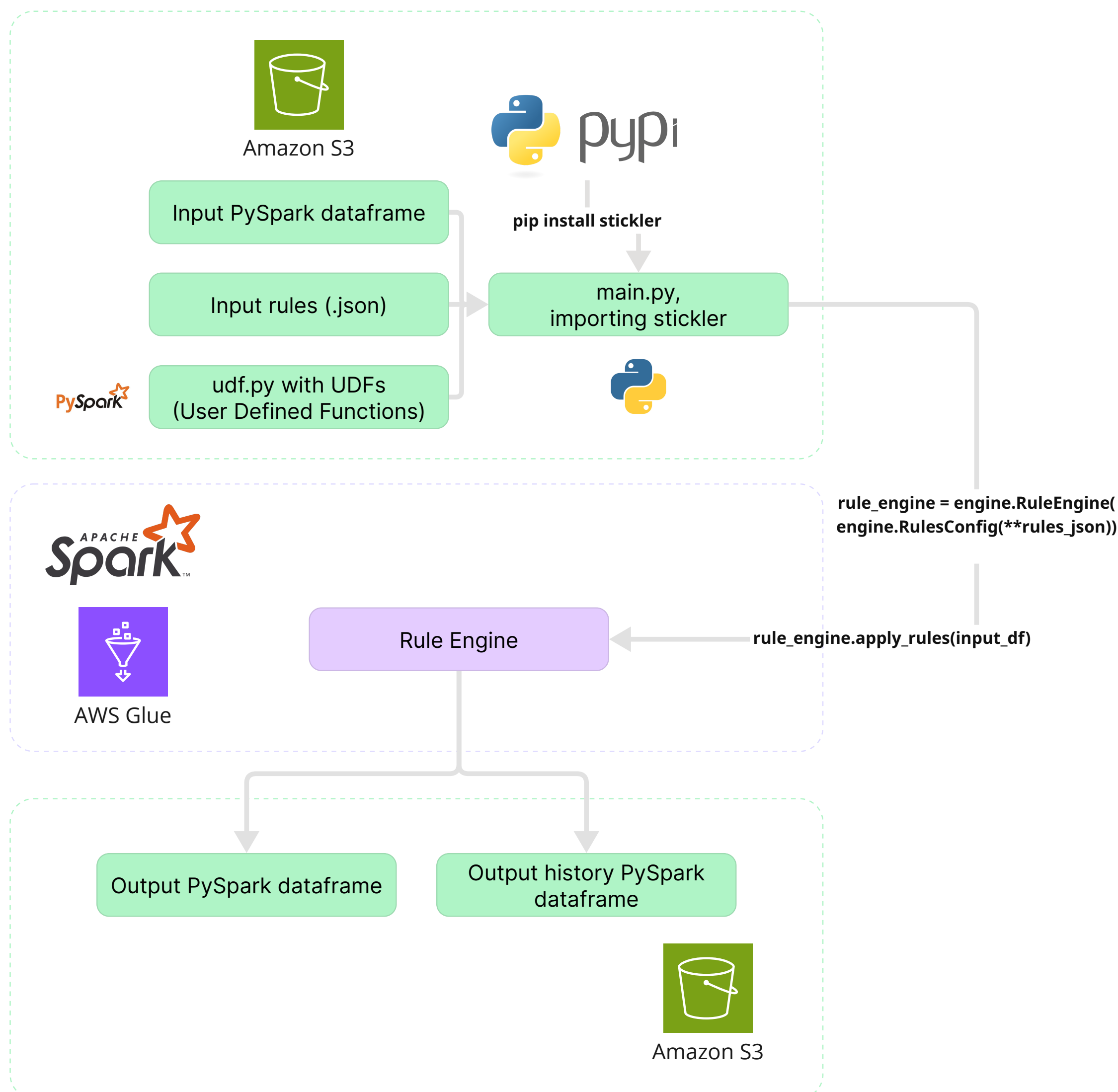

Figure 4.2: Architecture diagram.

The processing is orchestrated by a main Python script (e.g., `main.py`), which acts as the client for the rule engine library. The rule engine is packaged and distributed via PyPI (Python Package Index), allowing for straightforward installation using pip.

Within the `main.py` script, after importing the necessary modules from the `stickler` library, the rule engine is initialized. This involves parsing the JSON

rule configuration into an `engine.RulesConfig` object and then instantiating the `engine.RuleEngine`, assuming `engine` is the imported stickler module and `rules_json` is the loaded content of the rules configuration file:

Listing 4.4: Initializing the Rule Engine

```
import json
from stickler import engine


with open("rules.json") as f:
    rules_json = json.load(f)

rules_configuration = engine.RulesConfig(**rules_json)
rule_engine_instance = engine.RuleEngine(rules_configuration)
```

In the case of having UDFs, these need to be registered as such using `udf` from `pyspark.sql.functions`, and then passed to the engine as a dictionary. The UDFs should be defined in a separate Python file (e.g., `udf.py`), which is imported into the main script.

Listing 4.5: Defining UDFs in udf.py

```
from pyspark.sql.functions import udf
from pyspark.sql.types import StringType, IntegerType


def sample_udf_1(value):
   return value.upper()

def sample_udf_2(value):
   return len(value)

udfs_dict = {
   "to_upper": udf(sample_udf_1, StringType()),
   "string_length": udf(sample_udf_2, IntegerType()),
}
```

Listing 4.6: Initializing the Rule Engine with UDFs

```
import json
from stickler import engine
from udf import udfs_dict


with open("rules.json") as f:
    rules_json = json.load(f)

rules_configuration = engine.RulesConfig(**rules_json)
rule_engine_instance = engine.RuleEngine(rules_configuration, udfs=
   udf_dict)
```

In both cases, the core processing is triggered by invoking the `apply_rules` method on the instantiated engine, passing the input PySpark DataFrame:

Listing 4.7: Applying Rules to the DataFrame

```
output_df , history_df = rule_engine_instance.apply_rules(input_df)
```

Then, `stickler` Rule Engine internally utilizes Apache Spark's capabilities to execute these rules efficiently across distributed data.

Upon completion, the `apply_rules` method returns two **output PySpark DataFrames**:

1. **Output PySpark DataFrame**: This DataFrame contains the data transformed according to the applied rules.

2. **Output History PySpark DataFrame**: This DataFrame provides an audit trail, detailing which rules were applied to each row and what the value was at that moment of execution.

These resulting PySpark DataFrames can then be handled by the user as needed. For instance, they can be further analyzed, transformed or persisted to a storage system. In the AWS test environment, these output DataFrames are saved back to Amazon S3.

For the test implementation, the entire PySpark application is executed as a job on **AWS Glue**. AWS Glue provides a managed ETL (Extract, Transform, Load) service that can run PySpark scripts, making it a suitable environment for deploying and testing such a rule engine. Amazon S3 serves as the primary data lake: storing the input data, rule configurations, UDF files, and the resulting output and history data.

4.3. Core Components of the Rule Engine

The core of the rule engine is composed of several interrelated components that work together to parse, validate, execute, and track business rules. At the heart of the system is the rule parser and execution framework.

The rule engine will be explained in detail in the following sections, covering its main components and their interactions. This will be done following the natural flow of the rule engine's operation, starting from the initial parsing of rules, through the validation process, and finally to the rule application and execution.

The main components of the engine can be summarized as follows:

- **Rule parser**: This component is responsible for parsing the input JSON rule definitions, it's implemented through Pydantic. Ensures that the rules conform to the expected schema and structure.

- **Rule validation**: Validates the rules content against the input DataFrame, checks for potential issues such as naming errors before execution.

- **Rule execution**: As the core operational component, applies the validated rules to the input data. It iterates through each rule, evaluating its specified condition and, if the condition is met, executing the corresponding action. Also ensures that cascading effects are properly managed.

- **DataFrame**: The engine operates on a PySpark DataFrame (temporary), that is dynamically transformed based on the conditions evaluated and the actions executed by the applied rules.
- **Output**: Two outputs are generated: the transformed DataFrame and another one that serves as a history of the transformations done through the rule engine's execution.

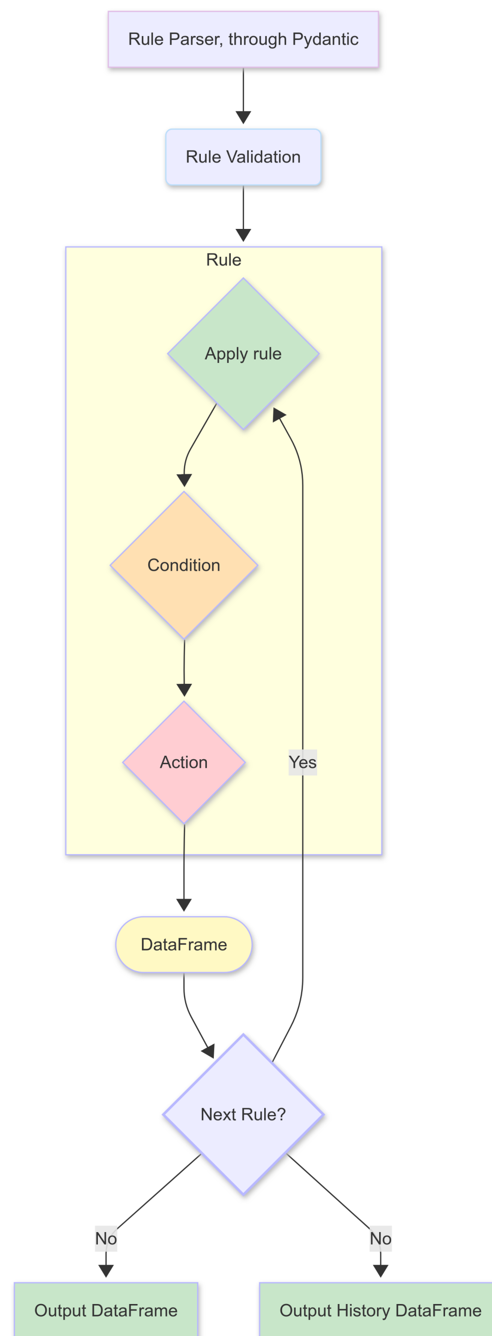

Figure 4.3: Core components of the Rule Engine.

4.3.1. Rule Parser and Validation

Rules are defined in an easy-to-understand JSON format, allowing engineers of varying expertise, to create and modify rules, as seen in Chapter 3 in Section 3.3.3. To parse these JSON rule definitions, the engine utilizes **Pydantic**. An example of a valid rule would be as follows:

Listing 4.8: Example of a Rule JSON

```
{
    "rules": [
        {
            "rule_name": "discountPrices",
            "execution_type": "accumulative",
            "conditions": {
                "expression": "stock > 0 AND price > 100"
            },
            "actions": {
                "output_col_name": "final_price",
                "expression": "price * 0.9"
            }
        }
    ]
}
```

Conditions and actions are expressed as SQL (*Structured Query Language*) expressions and are parsed using PySpark's native *expr* function. This integration allows users to utilize built-in SQL functions and reference other columns, whether they exist in the initial dataset or were dynamically created by earlier rules, directly within their rule definitions.

PySpark SQL is Spark's module for working with structured data using SQL queries or the DataFrame API. Unlike traditional SQL, which runs inside a single RDBMS (*Relational Database Management System*) instance, PySpark SQL compiles queries into distributed execution plans across a Spark cluster, enabling high-throughput processing of large datasets. It supports most ANSI SQL constructs, and lets you intermix SQL queries (`spark.sql("...")`) with DataFrame transformations (`df.select(...), df.filter(...)`) easily.

Once rules are parsed into Python objects, the engine performs an initial validation phase, in the following order:

1. **Name validator**: Checks that rule names are non-empty, unique and do not conflict with reserved names used internally (such as those for history tracking).

2. **Reference validator**: Ensures that all columns referenced in the conditions or actions exist in the input DataFrame or were generated by previously processed rules.

3. **Expression validator**: Attempts to execute the expressions on a dummy DataFrame (one that contains no rows but is built with the appropriate schema) to catch common errors like invalid operators or syntax issues.

Although these validations do not guarantee that all runtime issues will be caught, it successfully filters out many syntax and type-related errors.

The validators are implemented as a chain of responsibility, allowing each validator to handle its specific validation task. The implementation of this pattern will be discussed in more detail in the next section: Section 4.3.2.

The flow of the validation process can be seen in Figure 4.4.

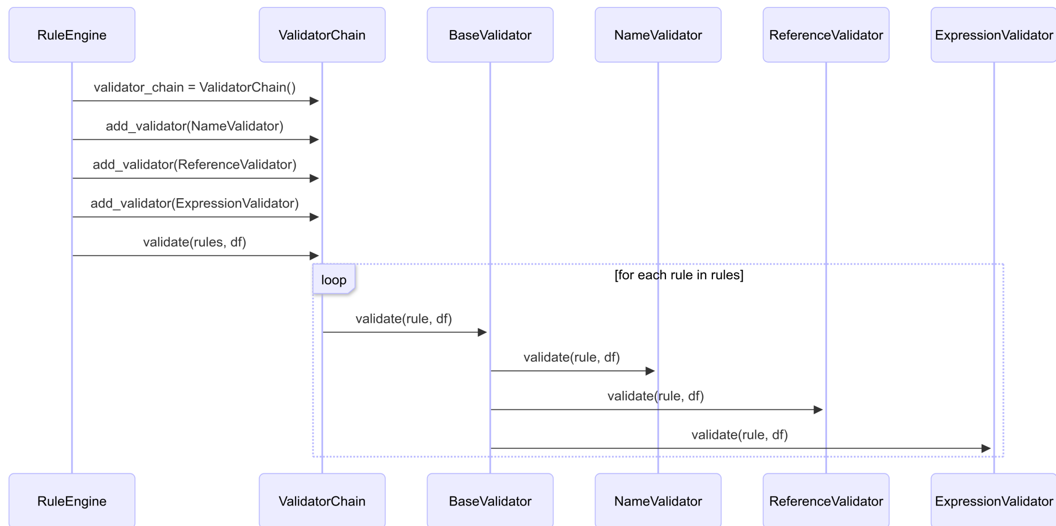

Figure 4.4: Rule validation sequence diagram.

4.3.2. Rule Application Strategies

The engine employs distinct strategies for applying rules, primarily determined by the existence of the **target output column**.

If this column does not yet exist, the engine first creates it. The new column is then populated based on the rule's logic. When a rule is processed for a given row, its action is applied if all specified conditions are met. This action typically involves computing a new value using the rule's **expression**.

If the conditions are not satisfied, the engine evaluates an "otherwise" clause, if one has been defined. In such cases, the expression within the "otherwise" clause determines the alternative value. If no "otherwise" clause is specified, the outcome depends on the column's prior state: existing values remain unchanged, while newly created columns (where the rule's main conditions failed) are set to a default of `None` (appropriately cast to the target data type).

Concurrently, all outcomes of rule applications are recorded in a **history DataFrame**.

Figure 4.5 illustrates the two main strategies for rule application, whether the target output column exists or not, and how the engine manages the assignment of values, including the handling of "otherwise" clauses and default values:

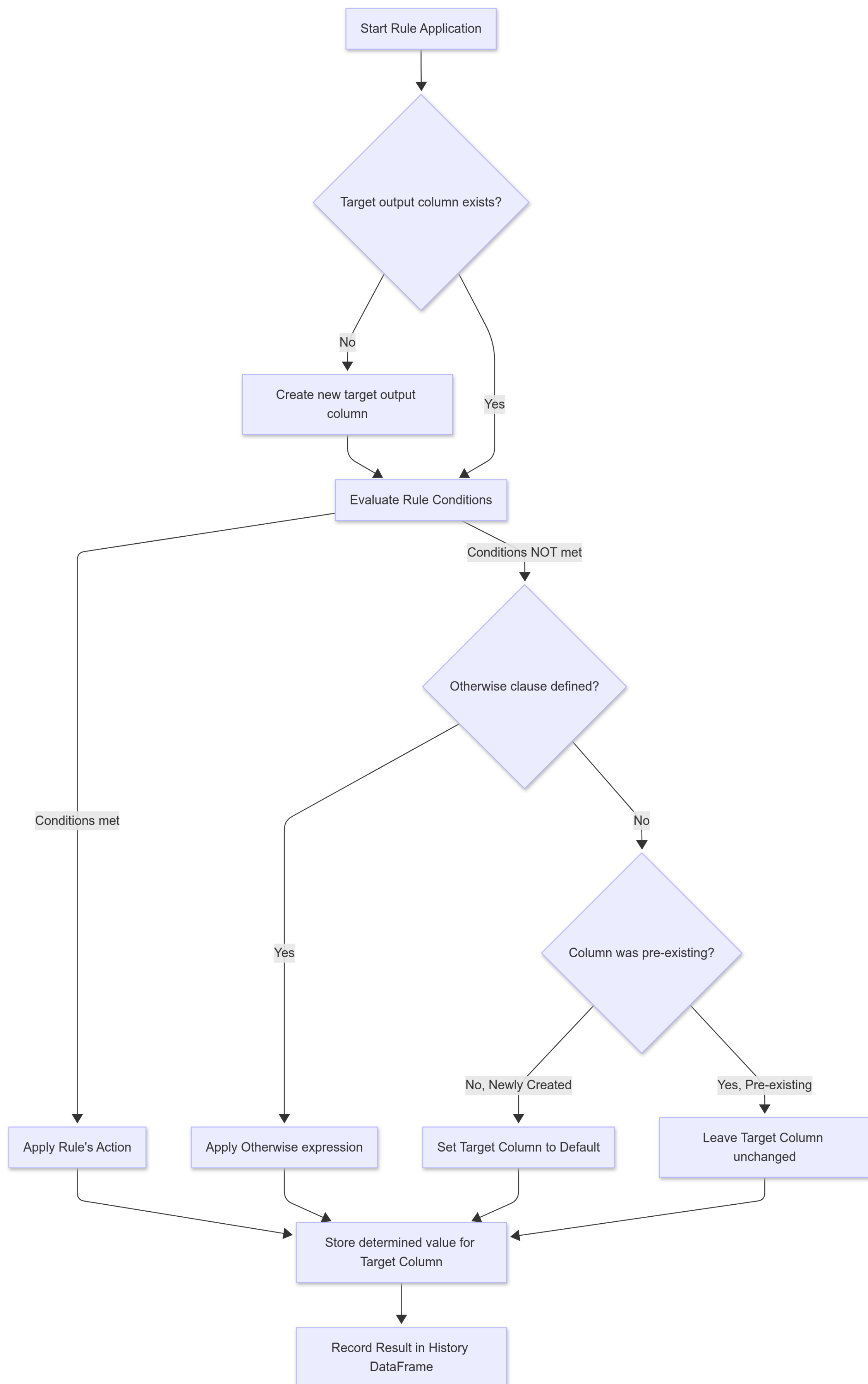

Figure 4.5: Rule application strategies diagram.

4.3.3. Execution Order, Dependencies and Cascading Rule Effects

While PySpark handles the underlying distributed data processing, rules within the engine are executed **sequentially** for each row, thereby eliminating dependency issues at the row level. That said, overall rule execution is parallelized across data partitions, meaning that independent rows can be processed concurrently.

The engine supports two execution models: **accumulative** and **cascading**. In the cascading model, rules are grouped into cascading groups, where a rule will only execute on a given row if no other rule within the same group has already been applied. To implement this, the engine temporarily stores the previous value in a dedicated column before applying a cascading rule. If the new value is blocked by the cascade condition, the engine restores the previous value from this temporary storage and then removes the auxiliary column. This design ensures that rules with dependencies or mutual exclusivity are applied in a controlled, deterministic manner.

As per Section 3.3.2's example, consider an online store applying multiple types of discounts using cascade groups. Each discount rule is assigned to one or more cascade groups to control their exclusivity during execution.

For instance, all **repeat customer discounts** can be placed in **cascade group 0**, while **seasonal discounts** belong to **cascade group 1**. Since these discounts target different contexts, they can be applied independently and simultaneously if both conditions are satisfied for a given customer.

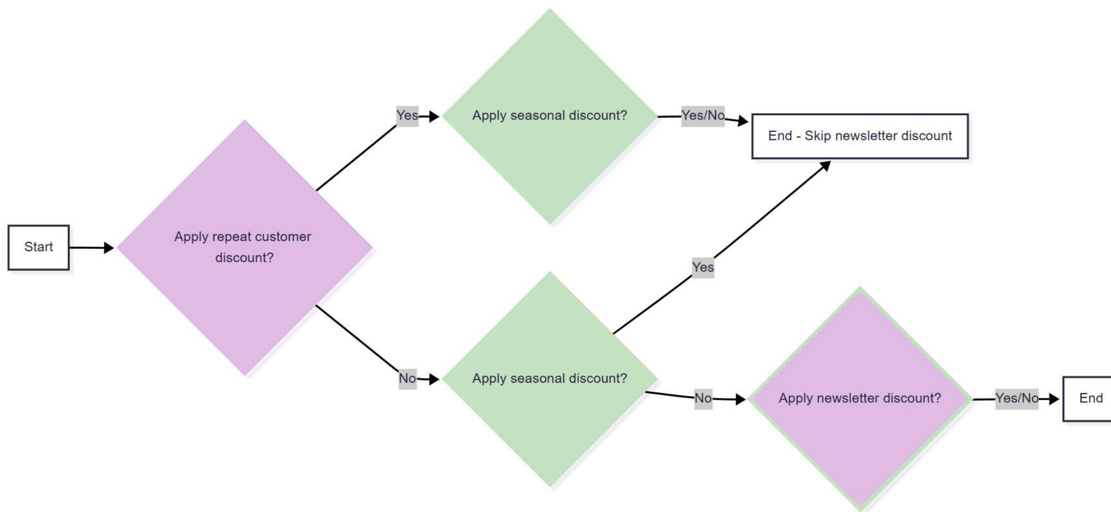

Figure 4.6: Discounts application diagram. Cascade group 0: Pink. Cascade group 1: Green.

Now consider a newsletter subscriber discount. This discount is intended to be exclusive: if a customer qualifies for any other discount, it should not be applied. To enforce this, **the rule is assigned to both group 0 and group 1**. Under the cascade execution model, a rule will only apply if no other rule in its group has already been executed on that row. By assigning the newsletter discount to both

groups, it becomes mutually exclusive with any discount in either category. This setup ensures precise control over rule interactions without hardcoding priorities, keeping the system flexible and maintainable.

If a rule's conditions are met but it is blocked due to cascade execution (i.e., another rule in the same group was already applied to that row), it will be marked as not executed in the history DataFrame. This is represented by a False value in the corresponding rule column, indicating that the rule was skipped due to group exclusivity despite matching the conditions.

4.3.4. History-Tracking Mechanism Implementation

To provide a transparent view of the entire rule execution process, the engine records a history of modifications. Rather than maintaining separate DataFrames throughout processing, which can be computationally expensive and prone to synchronization issues, the engine merges history information into a single unified DataFrame during execution. At the end of processing, this unified DataFrame is split into two: one containing the final computed values and another dedicated to the historical record of rule executions. Each time an action is applied, the resulting value is logged in the history DataFrame, allowing users to trace the evolution of data through the rule engine.

This split is performed without introducing row indexes because PySpark's operations inherently preserve row order, ensuring that the history aligns perfectly with the original data.

The structure and content of this historical record can be illustrated with an example, such as the `discountPrices` rule (detailed in Chapter 3, Section 3.3) applied to a product catalog. After rule execution, the dedicated history DataFrame would appear as in Table 4.1.

Product	**Stock**	**Price**	**discountPrices**	**discountPrices_Price**
Product A	50	90.00	False	90.00
Product B	20	135.00	True	135.00
Product C	10	108.00	True	108.00

Table 4.1: Example History DataFrame for the `discountPrices` rule (same as Table 3.3).

This history DataFrame includes the original input columns alongside additional columns specific to each rule's execution. As demonstrated, the rule `discountPrices` generates a boolean history column named after the rule itself, indicating whether the rule's conditions were met and its actions applied to each row. Furthermore, for each action defined within a rule that modifies or generates a column, an extra history column is created. This column is typically named by combining the rule name with the target output column name (`discountPrices_Price`), and it records the actual value produced by that rule's action for the specified output column.

4.4. Anatomy of the Implementation

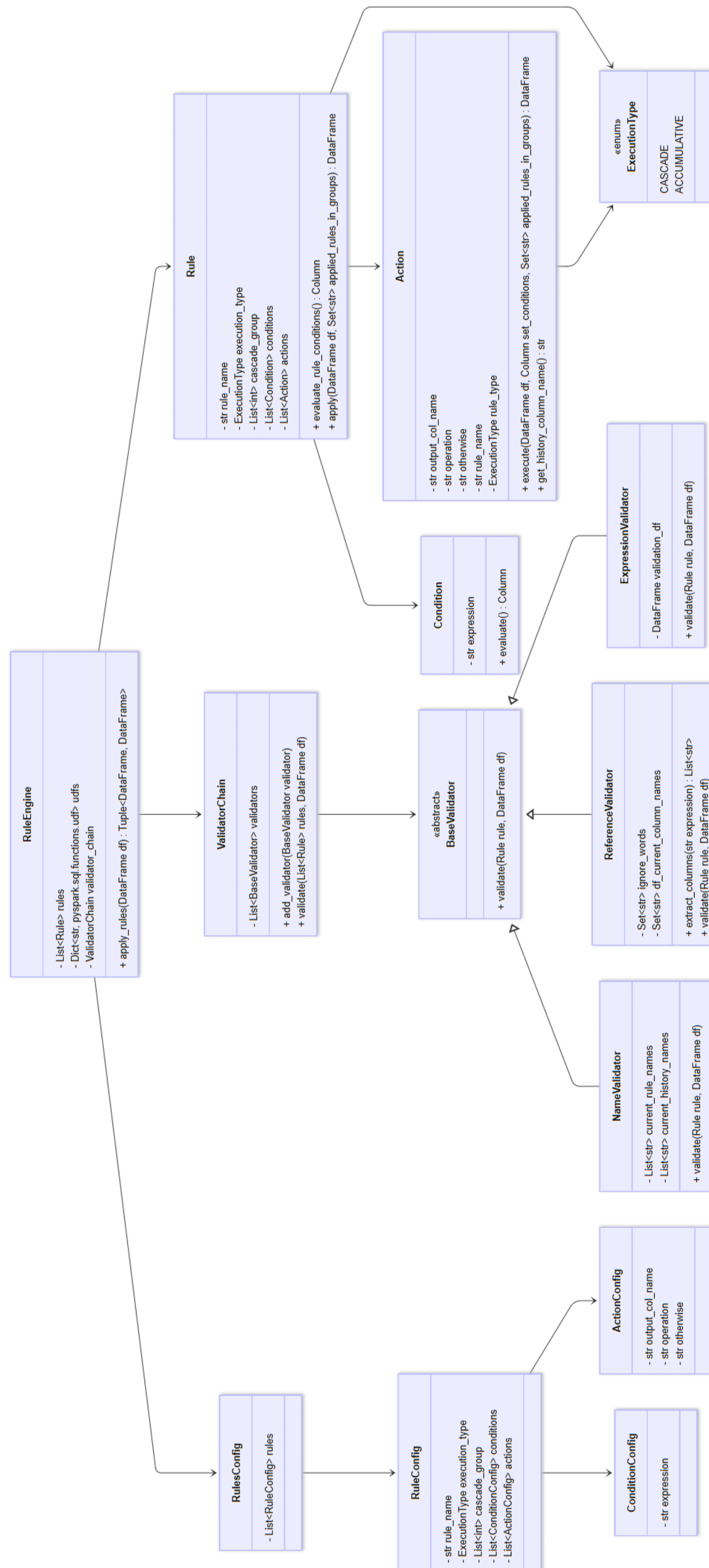

Figure 4.7: UML class diagram.

As shown in Figure 4.7, the *RuleEngine* class orchestrates the execution of the engine, acting as the central coordinator for processing user-defined rules. When instantiated, the engine first parses the incoming JSON configuration into strongly typed *RuleConfig* objects using Pydantic. This initial phase ensures that each rule's name, conditions, actions, and schema conform to expectations before any Spark logic is invoked. Although the rule expressions themselves are represented textually at this point, they are not evaluated until later. Instead, the engine simply verifies that they exist and meet the JSON schema requirements.

Once the rules are parsed, the engine registers any user-defined functions (UDFs) with Spark. These UDFs become available for later use inside Spark SQL expressions, allowing business users to reference custom logic. At the same time, the engine initializes a *ValidatorChain*, linking together validators like the *NameValidator*, *ReferenceValidator*, and *ExpressionValidator*. This chain embodies the Chain of Responsibility pattern: each validator inspects the rules and the input DataFrame in turn, catching schema mismatches, missing column references and basic expression errors.

With initialization complete, calling *apply_rules()* launches the main execution flow. The engine now runs the *ValidatorChain*. By performing this pre-flight checks, the engine minimizes runtime failures and provides clear, early feedback to the user if a rule is misconfigured. After passing validation, the engine proceeds to apply each rule sequentially, in the user-defined order. Depending on its configured *ExecutionType* (accumulative or cascade), a rule will evaluate its conditions using Spark's *expr* function and then execute one or more *Action* objects. Each action either creates or updates a column in the evolving DataFrame, and in cascade mode the engine selectively restores previous values when rules are blocked by earlier rules in the same cascade group.

Throughout this process, both the final results and a full audit trail are captured in a single, unified DataFrame. Output columns reflect the cumulative effect of all rule actions, while dedicated history columns-named using the pattern *ruleName_actionName*-record the intermediate values generated at each step. Finally, the engine splits this unified DataFrame into two: the Output DataFrame, containing the transformed data, and the History DataFrame, containing original columns plus one history column per executed action.

The overall flow and interaction of these components, from parsing and validation through to rule application and history generation, are visually detailed in the sequence diagram presented below (see Figure 4.8).

This design combines the Facade pattern (through the *RuleEngine*) and the Chain of Responsibility (for validation). It is worth noting, however, that the implementation does not strictly adhere to the definition of said patterns, as the engine is not a pure facade and the validators are not strictly a chain of responsibility. In spite of this, the current design is closely aligned with the principles of these patterns, achieving the intended goals of modularity, flexibility and maintainability.

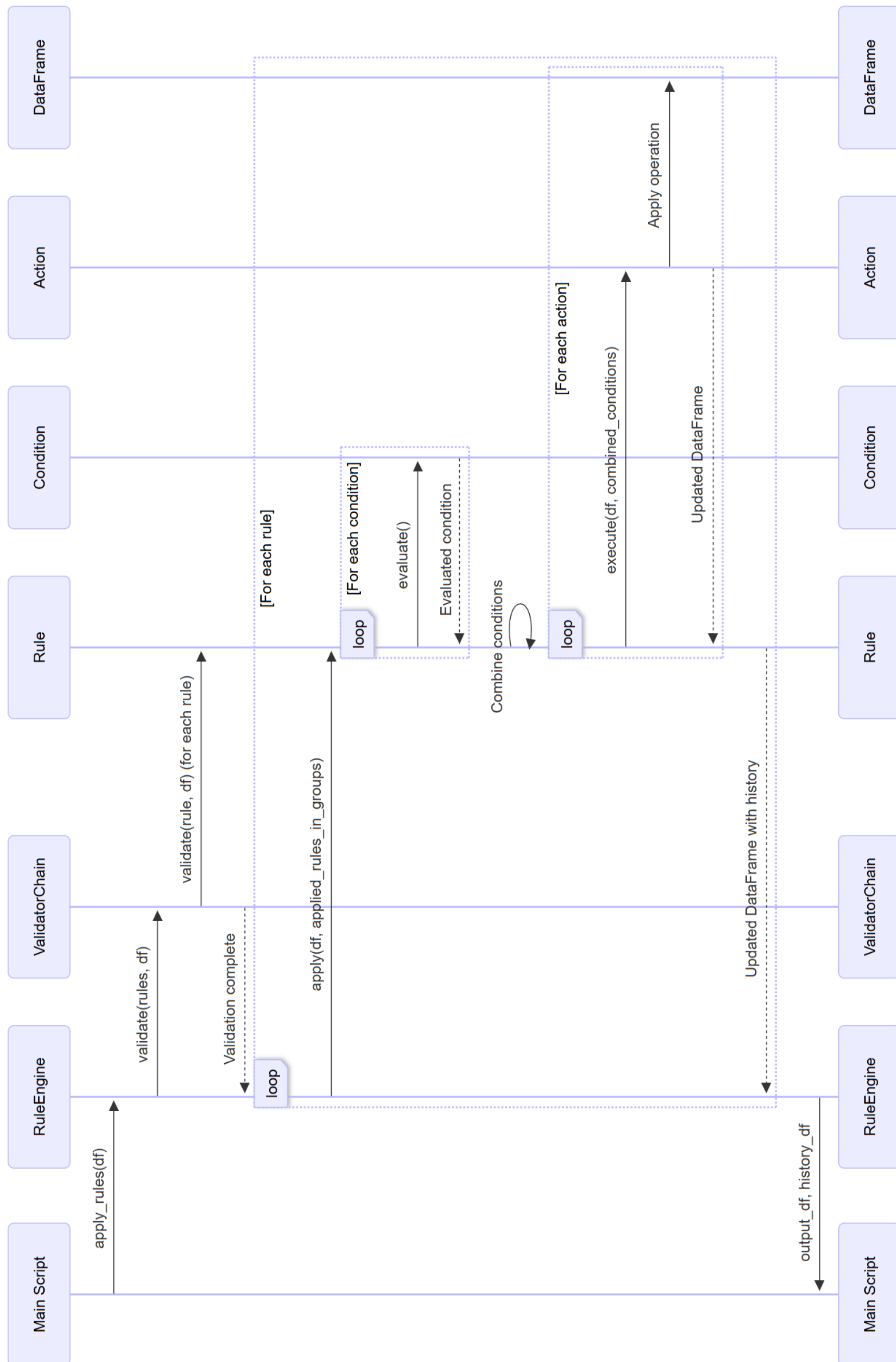

Figure 4.8: Sequence diagram.

A brief overview of the patterns used in the implementation is provided next:

Facade Pattern

The facade pattern is a structural design pattern that provides a simplified, unified interface to a more complex set of underlying subsystems or interfaces [57].

This pattern is applied on the *RuleEngine* class, which acts as the primary and simplified entry point. Instead of the user needing to worry about all the details of how rules are parsed, how validation works, or how the rules are applied to the DataFrame, they only need to call one function: *apply_rules()*. The engine takes care of setting everything up and running it in the right order behind the scenes. This makes it easier to use and keeps the complexity hidden.

Chain of Responsibility Pattern

The chain of responsibility pattern is a behavioral design pattern that allows a request to be passed along a chain of potential handlers until it is processed or the end of the chain is reached [56]. Each handler in the chain has the option to process the request or to delegate it to the next handler. This pattern promotes loose coupling between the sender of the request and its receivers.

The process flows as follows: a validator inspects the rules and data. If it finds no issues pertinent to its specific responsibility, it passes the request (implicitly, by allowing the chain to proceed) to the next validator in the sequence. If a validator detects an error (e.g., a missing column reference), the chain's defined behavior is to halt further processing and raise an error immediately.

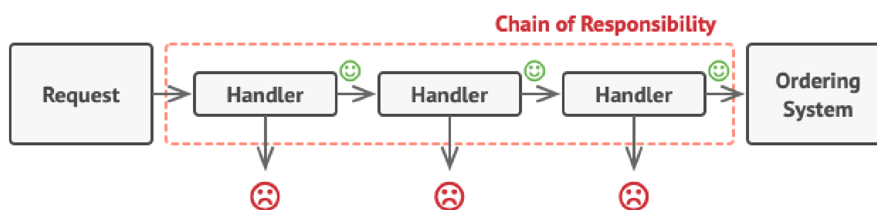

Figure 4.9: Chain responsability pattern.
By Refactoring Guru [56].

This pattern is applied on the validation system. There's a *ValidatorChain* class that contains a list of validators. When rules are validated, each validator gets a chance to check the rules one by one. If no errors are found, the validation keeps going. If something's wrong, the process stops and the error is raised. This setup makes it easy to add or remove validators later on without needing to change how the rest of the engine works.

4.5. Testing and Error Handling

To ensure the robustness and reliability of the rule engine, extensive testing and comprehensive error handling mechanisms have been integrated into the devel-

opment process. This chapter discusses the approach to unit testing, integration testing, and the error handling strategies implemented throughout the engine. The library used for testing is **pytest**, while the logging was implemented using the standard Python **logging** module.

4.5.1. Unit Testing

Unit tests have been developed to verify the functionality of individual components. Each class, such as the rule parser, validators, and execution framework, has been tested in isolation to ensure that it behaves as expected when provided with both valid and invalid inputs. For instance, tests for the *Expression Validator* verify that invalid SQL expressions or mismatched data types are correctly identified and that descriptive error messages are raised, guiding the user to the source of the error. Similarly, the *Reference Validator* is rigorously tested to ensure that any rule attempting to reference a non-existent column is caught early in the process.

To facilitate these isolated tests and manage the complex setup often required, the project utilizes pytest's fixture system. Fixtures are instrumental in providing a fixed, reliable baseline for tests, simplifying the creation and management of test-specific objects and resources. For example, a session-scoped fixture `fixture_spark` is used to initialize and manage the lifecycle of a Spark session for the entire test session, ensuring it is available when needed and properly stopped afterwards. Other common use cases include fixtures for generating consistent sample Spark DataFrames (e.g., `fixture_sample_df`) that rules can operate on, and fixtures that provide standardized or mock rule configuration objects, such as `fixture_sample_rule_config`. To promote reusability and maintain a clean test structure, fixtures shared across multiple test files are centralized in `conftest.py` files. pytest automatically discovers these `conftest.py` files and makes the defined fixtures available to any test within the same directory or its subdirectories, without requiring imports.

Furthermore, using Pydantic for schema validation, tests ensure that JSON rule definitions are correctly parsed into Python objects. These tests cover scenarios where the input may have missing fields or improperly formatted data, verifying that appropriate error messages are generated and that the engine rejects invalid rule definitions. Pydantic also allows for the definition of default values when a field is not defined. Each JSON definition is checked to ensure that default values are inserted when certain attributes are not provided. This guarantees that the system's behavior remains consistent even when not all values are explicitly defined by the user.

To run the tests, a `run-tests` command to the Makefile was added:

Listing 4.9: Makefile

```
.PHONY: code-check run-tests
code-check:
	isort .
	black .
	pylint src tests
run-tests:
	pytest
```

To ensure clarity and a consistent structure that reflects the behavior being tested, the tests within this project are written following the **Given-When-Then** convention [40]. This approach separates the test into three distinct sections:

- **Given**: Sets up the initial context or preconditions for the test.
- **When**: Executes the action or behavior being tested.
- **Then**: Asserts the expected outcome or state.

A summary of the battery of unit tests generated for the code is provided in the table below.

Table 4.2: Unit tests.

Test File	Test Case	Description
`test_action.py`	`config_creation`	Tests the creation and initialization of an action configuration.
`test_action.py`	`creation`	Tests the creation of an action object.
`test_action.py`	`execute_happy_path`	Tests the execution of an action under normal conditions.
`test_action.py`	`execute_no_otherwise`	Tests the execution of an action when no "otherwise" value is provided.
`test_action.py`	`execute_existing_column_with_otherwise`	Tests execution when the target column already exists and an "otherwise" value is given.
`test_action.py`	`execute_existing_column_no_otherwise`	Tests execution when the target column exists but no "otherwise" value is given.
`test_action.py`	`execute_cascade_single_rule_blocked`	Tests cascading behavior when a rule is blocked by a previous one.
`test_action.py`	`execute_cascade_single_rule_executed_fully`	Tests cascading when a rule is not blocked by any previous one.
`test_action.py`	`execute_cascade_single_rule_blocked_existing_column`	Tests cascading with a blocked rule and an existing column.
`test_action.py`	`execute_cascade_single_rule_executed_fully_existing_column`	Tests cascading with a not-blocked rule and an existing column.

test_action.py	execute_cascade_single_rule_executed_fully_existing_column_with_otherwise	Tests cascading with a fully executed rule, existing column, and "otherwise" value.
test_action.py	execute_cascade_multiple_rule_blocked	Tests cascading behavior when a rule is blocked by multiple previous ones.
test_action.py	execute_cascade_multiple_rule_executed_fully	Tests cascading when there are multiple previous rules, but don't block this one.
test_action.py	get_history_column_name	Tests the generation of the history column name.
test_action.py	str_representation	Tests the string representation of an action.
test_condition.py	evaluate	Tests the evaluation of a condition.
test_condition.py	str_representation	Tests the string representation of a condition.
test_engine.py	udfs_registration	Tests the registration of user-defined functions (UDFs).
test_engine.py	apply_rules	Tests the application of rules to a simple DataFrame.
test_expression_validator.py	single_rule_correct	Tests a single rule with a correct expression.
test_expression_validator.py	invalid_condition_expression	Tests an invalid condition expression.
test_expression_validator.py	invalid_action_expression	Tests an invalid action expression. It should raise an exception.
test_expression_validator.py	rule_with_new_column	Tests a rule that creates a new column (this is valid, and shouldn't raise an exception).
test_expression_validator.py	rule_with_new_column_invalid_expression	Tests a rule creating a new column with an invalid expression. It should raise an exception.
test_name_validator.py	single_rule_correct	Tests a single rule with a correct name.
test_name_validator.py	single_rule_empty_name	Tests a single rule with an empty name. It should raise an exception.
test_name_validator.py	single_rule_name_already_in_df	Tests a rule where the name already exists in the DataFrame. It should raise an exception.

test_name_validator.py	rule_name_duplicated	Tests for duplicated rule names. It should raise an exception.
test_name_validator.py	many_rules_correct	Tests multiple rules with correct names.
test_name_validator.py	single_rule_empty_output_col_name	Tests a single rule with an empty output column name. It should raise an exception.
test_name_validator.py	multiple_rule_duplicate_history_col_name	Tests rules that share a name with a history column name. It should raise an exception.
test_reference_validator.py	single_rule_correct	Tests a single rule with correct references.
test_reference_validator.py	extract_columns	Tests the extraction of column names from a rule.
test_reference_validator.py	non_existing_column	Tests a rule referencing a non-existing column. It should raise an exception.
test_reference_validator.py	column_defined_in_one_rule_and_used_in_another	Tests a column defined in one rule and used in another.
test_reference_validator.py	column_defined_in_one_action_and_used_in_another	Tests a column defined in one action and used in another.
test_reference_validator.py	condition_references_column_defined_in_same_rule	Tests a condition referencing a column defined in the same rule. It should raise an exception.
test_reference_validator.py	non_existing_udf_in_action	Tests a non-existing UDF in an action. It should raise an exception.
test_reference_validator.py	existing_udf_in_action	Tests an existing UDF in an action.
test_validator_chain.py	validator_chain	Tests the creation of the validator chain.
test_validator_chain.py	validators_in_chain	Tests that the validators, when added, are present in the chain.

4.5.2. Integration Testing

While unit tests validate isolated functionality, integration testing is crucial to ensure that the various components of the rule engine work together seamlessly within a PySpark workflow. Integration tests simulate real-world scenarios by applying a series of rules to sample datasets that reflect typical use cases.

An illustrative example of such an integration test is `test_discount_cascade_logic`.

In the *Given* phase of this test, a synthetic dataset is prepared, featuring diverse customer profiles. Each profile includes attributes such as an initial purchase price, the number of previous orders, the current season (e.g., spring, summer), and a flag indicating newsletter subscription status. Concurrently, a specific set of rules is defined in JSON format, including:

- A `RepeatCustomerDiscount`: This rule offers a 10% discount, is configured with an `accumulative` execution type, and belongs to `cascade_group: 0`. It applies if a customer has made previous orders.
- A `SeasonalDiscount`: This rule provides a 5% discount, is also `accumulative` (implicitly), assigned to `cascade_group: 1`, and is triggered if the season is "spring".
- A `NewsletterDiscount`: This rule grants a 15% discount and is configured with an `execution_type: cascade`. Crucially, it is associated with `cascade_group: [0, 1]`, meaning it will be blocked (i.e., not applied) if any rule from either cascade group 0 or cascade group 1 has already been successfully applied to the same data row.

During the *When* phase, the *RuleEngine* is invoked to process the input DataFrame against this defined rule configuration, applying the rules sequentially according to their definitions and cascade logic.

Finally, in the *Then* phase, the test asserts on both the resulting *Output DataFrame* and the *Output History DataFrame*. For the *Output DataFrame*, it validates that the final price for each customer accurately reflects the discounts applied (or blocked) as per the stipulated cascade rules. For example, it verifies that customers eligible for both repeat and seasonal discounts receive them (potentially in an order influenced by cascade group processing if prices are modified sequentially by group), and critically, that the `NewsletterDiscount` is not applied to these customers due to the cascade blocking condition. Conversely, for customers only eligible for the newsletter discount, it confirms this discount is correctly applied. The *Output History DataFrame* is also checked. Assertions here check that the recorded history accurately reflects which rules were triggered for each customer (e.g., boolean flags for `RepeatCustomerDiscount`, `SeasonalDiscount`, `NewsletterDiscount`) and logs any intermediate values as expected, such as the price generated by the `NewsletterDiscount_price` action.

4.5.3. Error Handling and Logging

Robust error handling is integral to the engine's design. Each validator is implemented to raise descriptive exceptions when encountering an error, such as invalid syntax, type mismatches, or missing references. These error messages are designed to pinpoint the exact rule and component where the error originated, thereby easing the debugging process. For instance, if the *Expression Validator* detects an error

in a rule's action expression, it will raise an exception with a message, pinpointing the rule's name and the problematic expression. These follow the format defined in Section 3.2.1.

Additionally, the engine incorporates logging mechanisms that record each step of the rule execution process. This is done through the **logging** native Python module. The logging not only aids in troubleshooting but also provides an audit trail for verifying how data is transformed across the execution pipeline. Detailed logs include information about rule validation outcomes, execution order, and any cascading conditions that may have blocked a rule's application. Also there are levels of logging, which can be configured to control the verbosity of the output. The levels include: `DEBUG`, `INFO`, `WARNING`, `ERROR` and `CRITICAL`.

One aspect worth mentioning about the logging system is the use of a **Null Handler**. This approach ensures that, by default, the engine's internal logger does not produce any output unless the user explicitly configures it. This is a common practice in well-designed Python libraries, such as Pandas, which follows the same approach. It gives end users full control over how logging is handled, allowing them to attach the engine's logger to their own logging configuration if desired, without interfering with existing logging behavior or cluttering the output unexpectedly. The default name for the logger is also defined in the same file, following the convention of using the package name as a prefix, i.e., `STICKLER`.

Listing 4.10: NullHandler Implementation

```python
# filepath: src/utils/logger.py
import logging
from logging import NullHandler

logger = logging.getLogger(f"STICKLER.{__name__}")
logger.addHandler(NullHandler())
```

Here are some examples of log entries that might be generated during different stages of rule processing:

- **Rule execution:**

```
DEBUG:STICKLER.rule:Applying rule: <Rule_Name>
DEBUG:STICKLER.rule:Rule <Rule_Name> applied successfully.
INFO:STICKLER.engine:Rule <Rule_Name>
 Conditions
 Condition1, Condition2, ...
 Actions
 Action1, Action2, ...
INFO:STICKLER.engine:All rules applied successfully.
```

- **Error handling (rule validation):**

```
ERROR:STICKLER.utils.validator.expression_validator:Error on
    rule '<Rule_Name>' definition: the expression '<
    Invalid_Expression>' is invalid. Please use a valid
    operator and syntax.
```

Chapter 5

Cloud Deployment and Performance

This chapter is dedicated to assessing the rule engine's performance in a realistic operational context. The evaluation methodology is designed to simulate demanding, real-world conditions, thereby providing a comprehensive understanding of the engine's scalability and accuracy when deployed within a cloud environment. The core of this evaluation revolves around a dynamic pricing scenario, specifically for flight tickets during a large-scale commercial event, like *Black Friday*. This scenario revolves around a complex set of rules to recalculate flight prices based on a multitude of factors, including flight routes, seat availability, demand surges, and various promotional discounts.

The subsequent sections will detail the construction of this test scenario, the characteristics of the dataset employed, the specific rules formulated to mimic dynamic pricing strategies, and the key metrics that will be measured to quantify the engine's performance.

5.1. Scenario Definition

The primary scenario for evaluating the rule engine's performance simulates a dynamic pricing system for a travel agency during an even of high demand, specifically during a commercial event similar to *Black Friday*. This simulated event is the travel agency's *Hot Week*, a week-long event featuring discounts on flight tickets. The rules won't only take into account discount promotions, but also surcharges for high-demand flights, which are common in the airline industry.

5.2. Dataset Description and Preparation

To ground the performance evaluation in a realistic context, a publicly available dataset containing flight price information was selected. The chosen dataset is the "Flight Price Prediction" dataset, sourced from *Kaggle*[1]. Kaggle is a well-regarded online platform that hosts a vast collection of datasets and serves as a community

[1] `https://www.kaggle.com/datasets/dilwong/flightprices`

hub for data scientists and machine learning practitioners to explore, share, and collaborate on data-driven projects [37].

This dataset comprises flight search results scraped from *Expedia*, a leading online travel agency. It consists of one-way flights found on the website between 2022-04-16 and 2022-10-05, weighing around 30GB. The size of the dataset is a significant factor in evaluating the rule engine's performance, as it allows for the simulation of a high-volume data processing environment.

The key columns from this dataset are:

- `legId` (*string*): An identifier for the specific flight leg.
- `searchDate` (*date*): The date (YYYY-MM-DD) when the flight data was recorded from Expedia.
- `flightDate` (*date*): The scheduled date (YYYY-MM-DD) of the flight.
- `startingAirport` (*string*): The three-character IATA code for the departure airport.
- `destinationAirport` (*string*): The three-character IATA code for the arrival airport.
- `fareBasisCode` (*string*): The fare basis code associated with the ticket.
- `travelDuration` (*string*): The total duration of the travel, typically represented in hours and minutes, for example, "PT5H30M". This format is based on the ISO 8601 duration format, where "PT" indicates a period of time, "5H" represents 5 hours, and "30M" represents 30 minutes.
- `elapsedDays` (*integer*): The number of days elapsed between the search date and the flight date. Usually 0.
- `isBasicEconomy` (*boolean*): A boolean flag indicating if the ticket corresponds to a basic economy fare.
- `isRefundable` (*boolean*): A boolean flag indicating if the ticket is refundable.
- `isNonStop` (*boolean*): A boolean flag indicating if the flight is non-stop.
- `baseFare` (*double*): The base price of the ticket in USD, before taxes and additional fees.
- `totalFare` (*double*): The total price of the ticket in USD, inclusive of taxes and other fees. This is the primary field targeted for recalculation by the pricing rules.
- `seatsRemaining` (*integer*): An integer representing the number of seats remaining on the flight at the time of search.
- `totalTravelDistance` (*integer*): The total travel distance in miles for the flight leg (may include missing values).

- `segmentsDepartureTimeEpochSeconds` (*string*): String containing the departure time (Unix timestamp) for each segment of the trip, concatenated by "||" for multi-segment journeys.

- `segmentsDepartureTimeRaw` (*string*): String containing the departure time (ISO 8601 format: YYYY-MM-DDThh:mm:ss.000±[hh]:00) for each segment, concatenated by "||" for multi-segment journeys.

- `segmentsArrivalTimeEpochSeconds` (*string*): String containing the arrival time (Unix timestamp) for each segment, concatenated by "||" for multi-segment journeys.

- `segmentsArrivalTimeRaw` (*string*): String containing the arrival time (ISO 8601 format: YYYY-MM-DDThh:mm:ss.000±[hh]:00) for each segment, concatenated by "||" for multi-segment journeys.

- `segmentsArrivalAirportCode` (*string*): String containing IATA codes for arrival airports of each segment, concatenated by "||" for multi-segment journeys.

- `segmentsDepartureAirportCode` (*string*): String containing IATA codes for departure airports of each segment, concatenated by "||" for multi-segment journeys.

- `segmentsAirlineName` (*string*): String containing the names of airlines operating each segment, concatenated by "||" for multi-segment journeys.

- `segmentsAirlineCode` (*string*): String containing two-letter codes for airlines operating each segment, concatenated by "||" for multi-segment journeys.

- `segmentsEquipmentDescription` (*string*): String describing the aircraft type for each segment, such as "Airbus A321", concatenated by "||" for multi-segment journeys.

- `segmentsDurationInSeconds` (*string*): String containing the duration in seconds for each flight segment, concatenated by "||" for multi-segment journeys.

- `segmentsDistance` (*string*): String containing the distance in miles for each flight segment, concatenated by "||" for multi-segment journeys.

- `segmentsCabinCode` (*string*): String indicating the cabin class for each segment, for example: "coach", concatenated by "||" for multi-segment journeys.

Since the dataset is already cleaned and pre-processed, no additional preparation steps were necessary. The dataset was directly loaded into Amazon S3 as a CSV file, the same format that was available in the original dataset in Kaggle.

5.3. Performance Evaluation Metrics

The performance evaluation of the rule engine will be based on four key metrics, running in different configurations to assess them: 2 workers, 5 workers and 10 workers for the first type of worker, and 5 workers for the second type. The cost of each worker is defined in DPUs, each DPU costing 0.44USD per hour at the time of writing. Two types of workers will be tested:

- **G.1X**: each worker of this type has 4 vCPUs and 16GB of memory. Each worker is equivalent to 1 DPU. This is the default worker type, for which it will be tested the configurations of 2, 5 and 10 workers.
- **G.2X**: each worker of this type has 8 vCPUs and 32GB of memory. Each worker is equivalent to 2 DPUs. This type of worker will be tested with 5 workers, which is equivalent to 10 DPUs.

The performance evaluations will be conducted using the aforementioned four specific configurations, which will be referenced throughout this chapter as follows:

- **Configuration 1**: This setup consists of **10 G.1X standard workers**. This configuration provides a total of 10 DPUs (40 vCPUs and 160 GB of memory).
- **Configuration 2**: This setup utilizes of **5 G.1X standard workers**, providing a total of 5 DPUs (20 vCPUs and 80 GB of memory).
- **Configuration 3**: This setup employs **2 G.1X standard workers**, resulting in a total of 2 DPUs (8 vCPUs and 32 GB of memory).
- **Configuration 4**: This setup uses **5 G.2X larger workers**. With each G.2X worker being equivalent to 2 DPUs, this configuration also provides a total of 10 DPUs (40 vCPUs and 160 GB of memory), similar to Configuration 1 but with resources concentrated in fewer, more powerful workers.

The metrics are as follows:

- **Execution time**: This metric measures the time taken by the rule engine to process a batch of flight data and apply the dynamic pricing rules. This includes the time for data loading (from S3), rule parsing and validation, application of all defined rules by the *Stickler* engine, and writing the output and history DataFrames back to S3. The job will be executed with different configurations, to assess the impact of different resource allocations on execution time.
- **Resource usage**: This metric monitors the resource consumption of the rule engine during the execution of the dynamic pricing rules. It includes CPU load and memory usage. The job will be executed with different configurations, to assess the impact of different resource allocations on performance.

- **Cost**: This metric evaluates the cost of executing the engine on AWS Glue, including the cost of data storage in S3 and the cost of the Glue job itself. The cost related to the AWS Glue job will be calculated for different configurations, to assess the impact of different resource allocations on cost.
- **Accuracy**: This metric assesses the correctness of the recalculated flight prices. Accuracy is essential to ensure that the dynamic pricing rules are functioning as intended and providing reliable results. The history DataFrame will be instrumental in this verification.

5.4. Rule Definition

The specific set of rules developed to instantiate the dynamic pricing strategy for the flight dataset, are detailed below. These rules aim to simulate a variety of pricing adjustments an airline might apply during a high-demand commercial event, primarily targeting the modification of the `totalFare` column. The rule set incorporates a mix of `ACCUMULATIVE` rules, where multiple adjustments can be applied sequentially, and `CASCADE` rules, where the application of one rule can prevent others within specified cascade groups from executing. The rules are organized into `cascade_groups` (0, 1, 2 and 3), which dictate the order of execution, and for `CASCADE` rules, define the blocking conditions.

The following ten rules constitute the dynamic pricing logic for this evaluation, which will be executed sequentially as indicated in Table 5.1.

Table 5.1: Summary of Rules for Performance Evaluation.

Rule Name	**Rule**	**Conditions**	**Actions on** `totalFare`	**Exec. Type**	**Casc. Group**
`Summer Economy Discount`	10% discount for summer basic economy flights with high seat availability.	`month(flightDate) IN (6,7,8), isBasicEconomy = true OR segmentsCabinCode LIKE '%coach%', seatsRemaining > 5`	Multiply by 0.90	`ACC`	1
`Last Minute Deal`	15% discount for flights booked $\leq$ 3 days before departure.	`datediff(flightDate, searchDate) <= 3`	Multiply by 0.85	`CASC`	1

Continued on next page

Table 5.1: Summary of Rules (continued from previous page).

Rule Name	**Rule**	**Conditions**	**Actions on `totalFare`**	**Exec. Type**	**Casc. Group**
`Advance Booking Premium`	10% premium for flights booked $\geq$ 60 days in advance.	`datediff(flightDate, searchDate) >= 60`	Multiply by 1.10	`CASC`	1
`Refundable Ticket Surcharge`	Adds $20 surcharge for refundable tickets.	`isRefundable = true`	Add $20	`ACC`	0
`Airport Hub Discount`	3% discount for flights departing from major hubs (ATL, DFW, ORD, LAX).	`startingAirport IN ('ATL','DFW', 'ORD','LAX')`	Multiply by 0.97	`ACC`	0
`High Demand Surcharge`	15% surcharge if seats remaining $<$ 5.	`seatsRemaining < 5`	Multiply by 1.15	`ACC`	2
`Off PeakDay Discount`	8% discount for flights on off-peak days (Tuesday, Wednesday, Thursday).	`dayofweek (flightDate) IN (3,4,5)`	Multiply by 0.92	`CASC`	2
`LongHaul Business Discount`	$50 discount for long-haul ($>$2500 miles) business class tickets.	`segmentsCabinCode LIKE '%business%', totalTravelDistance > 2500`	Subtract $50	`ACC`	0

Continued on next page

Table 5.1: Summary of Rules (continued from previous page).

Rule Name	**Rule**	**Conditions**	**Actions on totalFare**	**Exec. Type**	**Casc. Group**
Extended Layover Discount	7% discount for flights with >1.5h layover and >5 seats remaining. Creates layover_hours column. Uses a UDF in the condition: max_layover_hours.	Multi-segment flight: size(split(segmentsDepartureTimeEpochSeconds, '||')) > 1, max_layover_hours() > 1.5, seatsRemaining > 5	Multiply by 0.93; New layover_hours column, stores the result of the layover hours there are between all the segments. Otherwise, it's 0.	ACC	3
RedEye SeatSale	Special discount of 12% applied via UDF (apply_redeye_discount) for basic economy "red-eye" flights if the flight costs more than $50. Refers to flights occuring during the night, where passangers don't obtain a full night's sleep.	isBasicEconomy = True OR segmentsCabinCode LIKE '%coach%', totalFare > 50.	Set to UDF result; apply_redeye_discount	CASC	[1,3]

The full JSON representation of these rules is provided in Appendix A.

Two User Defined Function (UDF) are included in the rule set. One is a simple UDF that calculates the maximum layover hours between segments of a multi-

segment flight, `max_layover_hours()`, and it is implemented in the condition. The other is a more complex UDF that applies a discount of 12% (action) when the flight is of type “red-eye”. This happens when the flight departs before 6:00 AM and after 00:00AM. If not, it does not apply any discount. This UDF is implemented in the action of the rule `RedEyeSeatSale` and is called `apply_redeye_discount()`. Specifically, this UDF is implemented as a mix of a condition and an action. This is done to showcase another allowed case in the rule engine.

5.5. Set up

As discussed in Chapter 4, the rule engine requires a `main.py` file to serve as the entry point for the AWS Glue job, apart from the JSON configuration file for the rules. Additionally, User Defined Functions (UDFs) can be included in a separate file.

However, to keep the test simple, the UDFs were included in the same file as the main function, as well as the rules. This is not a recommended practice for production, where each file should be stored separately, and retrieved from a bucket or similar.

The engine also needs an input dataset, which was previously uploaded to an Amazon S3 bucket, in its original format (CSV).

The script used to run on AWS Glue can be found on Appendix B. Keep in mind that it contains dependencies that are only found in an AWS Glue development environment, which can be obtained by either developing directly on the cloud, or setting up a Docker container with AWS Glue, which allows to test locally.

The AWS Glue ETL job configuration is straight-forward, requiring no other configuration than adding arguments passed to the program, and setting *Stickler* as one of its *pip*-dependencies. The number and power of workers can be set from the same menu.

5.6. Results

This section presents the results of the performance evaluation, structured around the four key metrics outlined earlier: execution time, resource usage, cost and accuracy. For the first three metrics, the results are presented in a comparative format, since they are all related to the performance of the engine in different configurations: 2 workers, 5 workers and 10 workers for the first type of worker, and 5 workers for the second type.

5.6.1. Execution time

The execution times for the different configurations, were as follows:

- **Configuration 1 (10 standard workers):** Utilizing 10 standard workers (total of 40 vCPUs, 160 GB RAM, 10 DPUs), the job completed in **8 minutes and 5 seconds** (485 seconds).

- **Configuration 2 (5 standard workers):** With 5 standard workers (total of 20 vCPUs, 80 GB RAM, 5 DPUs), the execution time increased to **14 minutes and 17 seconds** (857 seconds). Halving the resources from Configuration 1 roughly doubled the execution time, indicating a near-linear scalability in this range for this particular workload.

- **Configuration 3 (2 standard workers):** Further reducing the resources to 2 standard workers (total of 8 vCPUs, 32 GB RAM, 2 DPUs), the job duration extended considerably to **51 minutes and 33 seconds** (3093 seconds). This demonstrates a more significant performance degradation, suggesting that below a certain threshold of parallelism, overheads may have a more pronounced impact.

- **Configuration 4 (5 larger workers):** Employing 5 larger workers, each with 8 vCPUs and 32 GB RAM (total of 40 vCPUs, 160 GB RAM, 10 DPUs), the execution completed in **8 minutes and 45 seconds** (525 seconds). This configuration offers the same total DPU count and vCPU/RAM resources as Configuration 1, but distributed among fewer, more powerful workers. The execution time is comparable to Configuration 1, though slightly longer, which might suggest per-worker overhead for this specific job.

These execution times are compared in Figure 5.1 shown belown.

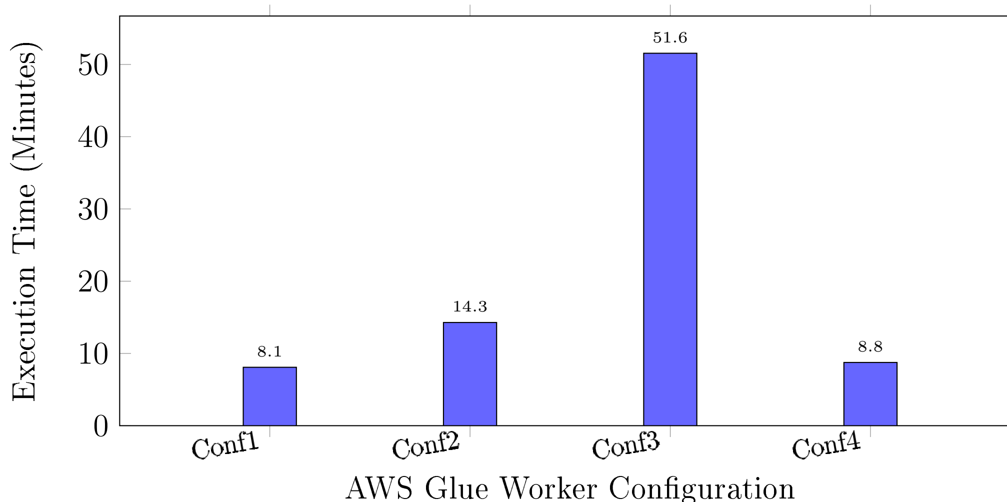

Figure 5.1: Comparison of execution time for different AWS Glue worker configurations.

5.6.2. Resource usage

The following discussion is based on monitoring data from AWS Glue, with illustrative graphs provided for Configuration 1 (10 G.1X standard workers), as seen below. According to observational data, the general patterns of CPU and memory utilization were found to be broadly similar across the other tested configurations

when adjusted for the scale of parallelism, indicating consistent behavior of the application under varying worker counts.

CPU load (%)

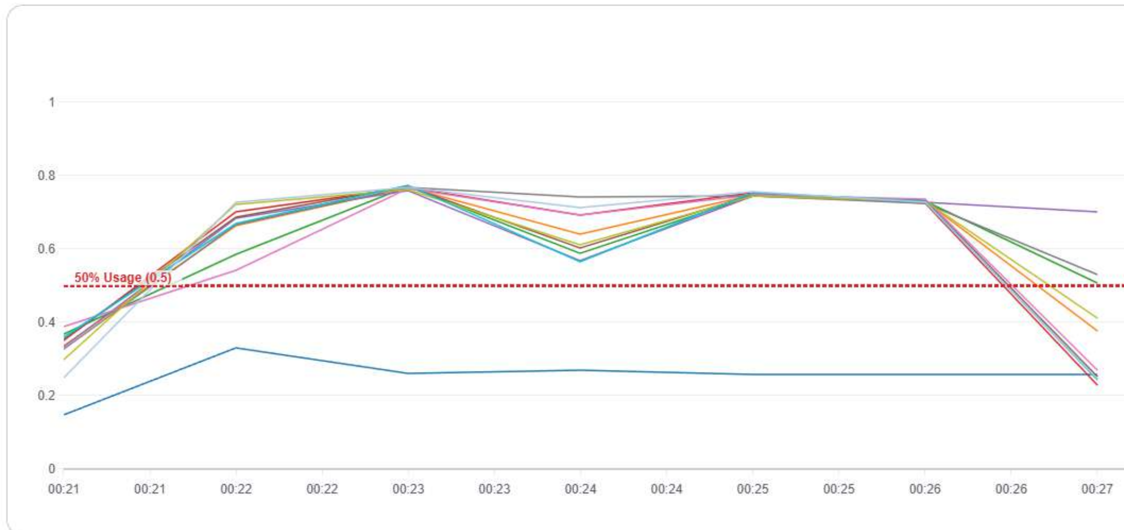

Figure 5.2: CPU load for the first configuration (10 workers).

Memory profile: drivers and executors (%)

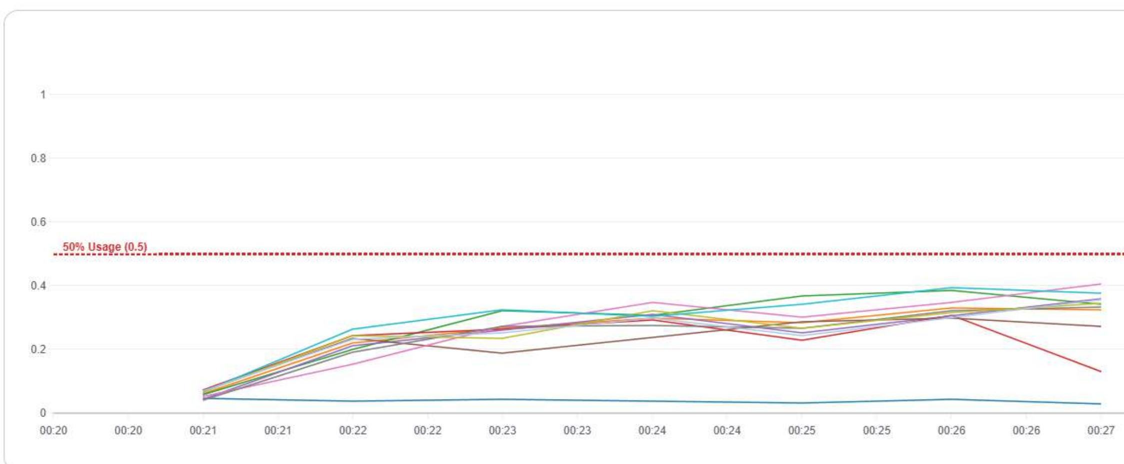

Figure 5.3: Memory profile for the first configuration (10 workers).

As observed in Figure 5.2, the CPU load exhibits a characteristic behavior for a Spark batch job. There is an initial ramp-up period (around 00:21 to 00:22 in the depicted segment) where tasks are distributed and executors begin active processing. During the main processing phase (approximately 00:22:30 to 00:25:30), most executors show a sustained high CPU utilization, generally fluctuating between 60% and 80%. This level of activity suggests that the workers are actively engaged in computation, efficiently utilizing the allocated vCPUs. One line, consistently lower

(around 25-35%), likely represents the Spark driver or a less burdened executor, which is typical as the driver orchestrates tasks rather than performing the bulk of the data processing itself. Towards the end of the job segment shown (after 00:26:00), CPU utilization across executors declines as tasks complete and resources are wound down. Most executors operate well above the 50% utilization mark during peak processing and indicates good parallelism and effective use of CPU resources for the rule application workload.

Regarding the memory profile, this illustrates how memory is consumed by the Spark driver and executors throughout the job's lifecycle. Figure 5.3 shows this profile for Configuration 1. Each G.1X worker in this configuration is allocated 16 GB of RAM. The memory usage pattern, demonstrates stable and well-managed memory consumption. After an initial phase where memory is allocated and data structures are initialized (around 00:21), the memory utilization for most executors gradually increases and then stabilizes, generally remaining within the 20% to 40% range of their allocated 16 GB. Similar to the CPU profile, one line (likely the driver) maintains a very low and stable memory footprint (around 3-5%).

The memory usage for all components stays comfortably below the 50% threshold throughout the observed period, and significantly below critical levels that might indicate memory pressure or risk of `OutOfMemoryErrors`. This suggests that the 16 GB of RAM per G.1X worker (and by extension, the 32 GB per G.2X worker in Configuration 4) is adequate for the dataset size and the complexity of the operations performed by the rule engine, including data shuffling and the storage of intermediate data. The absence of sharp spikes or sustained high memory usage points towards efficient memory management by PySpark and the rule engine's implementation for this workload.

5.6.3. Cost

The total cost of running the rule engine consists of two main components: data storage costs on Amazon S3, which remain constant across all configurations, and computational costs associated with AWS Glue job execution, which vary depending on the selected worker configuration.

Storage costs

The data involved in a single batch job includes the original input dataset and the two generated output DataFrames (final transformed data and history). The approximate sizes are:

- **Original input DataFrame**: 30.0 GB
- **Output DataFrame**: 6.8 GB
- **Output history DataFrame**: 8.4 GB

The total storage for these datasets amounts to $30.0 + 6.8 + 8.4 = \mathbf{45.2\ GB}$. Assuming an Amazon S3 storage cost of $0.023 per GB per month (this pricing may

vary by region, in this case it was done with `eu-central-1`), the monthly cost for persistently storing this data would be:

$$45.2 \text{ GB} \times \$0.023/\text{GB/month} = \$1.0396 \text{ per month}$$

Execution Costs

The AWS Glue execution costs are determined by the number of Data Processing Units (DPUs) used and the duration of the job, billed per DPU-hour. The standard AWS Glue worker (G.1X type: 4 vCPU, 16 GB RAM) corresponds to 1 DPU, and the larger worker (G.2X type: 8 vCPU, 32 GB RAM) corresponds to 2 DPUs. The cost per DPU-hour at the moment of the evaluation is $0.44 in `eu-central-1`, but this may vary by region.

The execution costs for the different configurations tested were as follows:

- **Configuration 1 (10 standard workers, 10 DPUs):** Job duration of 8 minutes 5 seconds (approximately 0.1347 hours). Cost = 10 DPUs$\times$0.1347 hr$\times$ \$0.44/DPU-hr $\approx$ \$**0.59**.

- **Configuration 2 (5 standard workers, 5 DPUs):** Job duration of 14 minutes 17 seconds (approximately 0.2381 hours). Cost = 5 DPUs $\times$ 0.2381 hr $\times$ \$0.44/DPU-hr $\approx$ \$**0.52**.

- **Configuration 3 (2 standard workers, 2 DPUs):** Job duration of 51 minutes 33 seconds (approximately 0.8592 hours). Cost = 2 DPUs $\times$ 0.8592 hr $\times$ \$0.44/DPU-hr $\approx$ \$**0.76**.

- **Configuration 4 (5 larger workers, 10 DPUs):** Job duration of 8 minutes 45 seconds (approximately 0.1458 hours). Cost = 10 DPUs $\times$ 0.1458 hr $\times$ \$0.44/DPU-hr $\approx$ \$**0.64**.

These figures illustrate that while reducing the number of workers (DPUs) initially decreases the per-hour cost rate, the extended execution time can lead to a higher overall job cost if the total DPU-hours increase significantly, as seen in Configuration 3, but not always. For example, Configuration 2 (5 standard workers) offered the lowest execution cost among the tested scenarios for this specific workload, despite not being the fastest. Configuration 4, using larger workers but the same total DPU count as Configuration 1, resulted in a slightly higher cost due to its marginally longer execution time. This can be visualized in Figure 5.4.

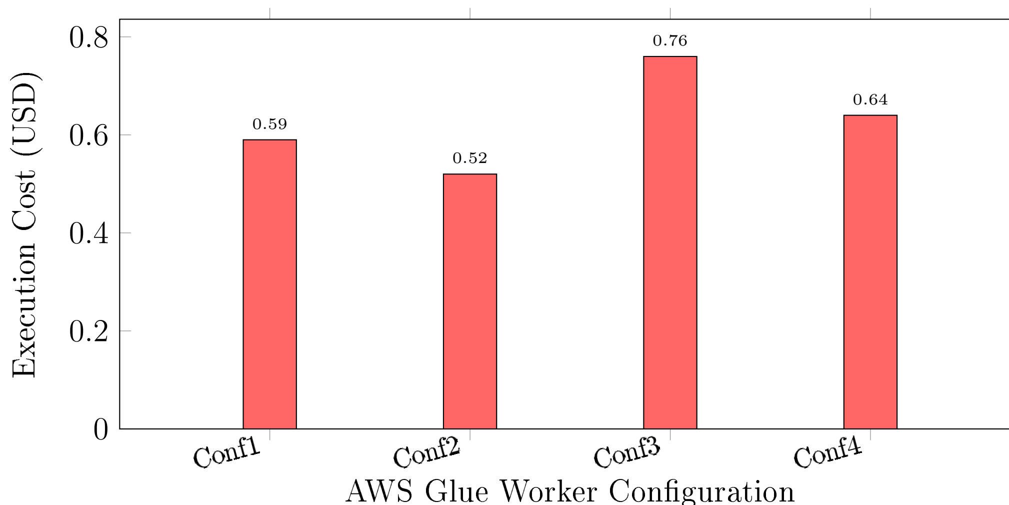

Figure 5.4: Comparison of execution cost for different AWS Glue Worker configurations.

5.6.4. Accuracy

In this section, a few examples will be provided, showing the correct execution of rules.

5.6.4.1. Example 1

Table 5.2 highlights how the rule engine evaluated a single flight record, combining both accumulative and cascade logic. It begins with the raw flight details: flight date, route, fare, and layover, and then show, for each rule, whether its conditions were met. In this example, only two rules fired. The *LastMinuteDeal* applied first, because the flight date (2022-04-17) and search date (2022-04-16) differ by just one day. This 15% discount reduced the fare from $204.97 to $174.22. Next, the *AirportHubDiscount* kicked in since the flight originated at ATL, one of the designated hub airports, taking the previously discounted fare down by another 3% to approximately $168.05.

All other rules, such as the extended layover, summer economy, or red-eye discounts, did not apply, either because their conditions were not met (a layover of 0 hours, non-economy cabin, or mid-day departure) or because the flight did not match the rule's criteria. Marking these as `False` in the history DataFrame confirms that the engine correctly skipped them. It can be seen that each rule fires only when intended and that accumulative discounts stack in the proper sequence.

Field	**Value**
legId	9ca0e81111c683bec1012473feefd28f
searchDate	2022-04-16
flightDate	2022-04-17
Route	ATL to BOS
isBasicEconomy	False
isRefundable	False
isNonStop	True
seatsRemaining	9
baseFare	$217.67
totalFare (pre-rules)	$204.97
layover_hours	0.0
Rule Results	
SummerEconomyDiscount	False
LastMinuteDeal	True → $204.97 x 0.85 = $174.22
AdvanceBookingPremium	False
RefundableTicketSurcharge	False
AirportHubDiscount	True → $174.22 x 0.97 = $168.05
HighDemandSurcharge	False
OffPeakDayDiscount	False
LongHaulBusinessPremium	False
ExtendedLayoverDiscount	False
RedEyeSeatSale	False

Table 5.2: Flight row with applied rules and final fare.

5.6.4.2. Example 2

In this scenario, the only rule that fires is *HighDemandSurcharge*, since just two seats remain on a multi-leg flight on 2022-04-19, triggering a 15% surcharge that pushes the fare from $372.24 up to about $428.09. Although the day of week (Tuesday) qualifies for the *OffPeakDayDiscount*, that rule belongs to the same cascade group as *HighDemandSurcharge*. Under cascade logic, once a rule in the group has executed for a row, any subsequent rule in the same group is blocked, hence *OffPeakDayDiscount* is skipped and marked `False`. All other rules also did not meet their conditions. This confirms that cascade blocking works correctly: the engine applies only the highest-priority surcharge per group and prevents conflicting discounts from stacking.

Field	Value
legId	c33fc9d40e2f8b4df6d27ae77f06ae82
flightDate	2022-04-19
Route	ATL → EWR → CLT
isBasicEconomy	False
isRefundable	False
isNonStop	False
seatsRemaining	2
totalFare (after prior rules)	$372.24
Day of Week	Tuesday
Layover hours	0.0
Rule Results	
HighDemandSurcharge	True → $372.24 x 1.15 = $428.09
OffPeakDayDiscount	False (cascade-blocked by *HighDemand*)
(other rules)	did not apply (all False)

Table 5.3: Flight row where *HighDemandSurcharge* applied and *OffPeakDayDiscount* was blocked.

5.6.4.3. Example 3

In this case, two rules fired in sequence for the flight on 2022-04-17 (ATL → EWR → BOS). First, the *LastMinuteDeal* applied because the ticket was searched just one day before departure, reducing the fare by 15%. Next, the *ExtendedLayoverDiscount* triggered: the itinerary includes an overnight layover of approximately 6.3 hours, exceeding the 1.5 hour threshold. This rule then applies a further 7% discount to the already reduced fare. The combined effect perfectly matches the values recorded in the history DataFrame, confirming that the engine correctly chains both accumulative and cascade rules and calculates layover durations as intended.

Field	Value
legId	a6a69aee2e8f75f9b12a5f6b60e65b59
flightDate	2022-04-17
Route	ATL → EWR → BOS
isBasicEconomy	True
seatsRemaining	9
totalFare (pre-rules)	$253.29
Layover hours	6.3
Rule Results	
LastMinuteDeal	True → $253.29 x 0.85 = $215.30
ExtendedLayoverDiscount	True → $215.30 x 0.93 = $200.61
(other rules)	did not apply (all False)

Table 5.4: Flight row where *ExtendedLayoverDiscount* applied.

Chapter 6

Conclusions and Future Work

6.1. Conclusions

This project focused on the development and evaluation of *Stickler*, an open-source business rule engine designed for PySpark, particularly for cloud-based big data environments. The primary motivation was to address the scarcity of user-friendly, open-source tools capable of integrating business rule execution directly into Spark workflows.

Stickler substantially met its defined objectives. The engine is designed for scalability, enabling it to process large datasets efficiently by leveraging Spark's distributed computing capabilities. Performance evaluations on AWS Glue using a significant flight dataset demonstrated effective resource utilization and improved execution times with increased worker configurations, confirming its suitability for big data tasks. Furthermore, *Stickler* is intended for broad applicability; rules are defined in a structured JSON format, and the project being *pip*-installable, makes it accessible for users with varying levels of data engineering expertise to implement it in their pipelines. The support for custom User-Defined Functions (UDFs) offers even more versatility, allowing for the incorporation of very specific business logic.

Regarding the rule engine's implementation on cloud environments, its availability on PyPI facilitates straightforward installation and adoption.

Stickler provides robust control over rule execution. It supports distinct execution types: "accumulative" for additive rule effects and "cascade" for conditional execution within defined groups, offering flexibility for complex scenarios. Another key feature is its integrated history tracking mechanism. This system records all data modifications during rule execution into a dedicated history DataFrame. This ensures transparency, auditability, and traceability, which are valuable for validation, debugging and compliance.

The rule engine addresses several common limitations found in existing open-source rule engines, particularly concerning their integration with Spark for large-scale data processing. Unlike many traditional tools that may struggle with distributed data or lack native Spark compatibility, *Stickler* is developed specifically for PySpark. Prior to execution, *Stickler* employs a comprehensive validation system, based on a chain of responsibility pattern, to verify rule definitions and preemptively

identify issues related to naming, column references, or expression syntax.

Performance was a key consideration in its implementation, and its successful deployment and testing on AWS Glue, using a flight pricing scenario, confirmed its suitability for big data use cases. The results showed effective CPU and memory utilization, with execution times scaling appropriately with allocated resources. The cost analysis also indicated that configurations can be optimized to balance performance and expenditure for given workloads.

In summary, *Stickler* fills an important gap by offering an open-source, PySpark-native rule engine that is both reliable and easy to scale. It brings clear, auditable decision logic directly into distributed data pipelines, letting teams define and manage complex business rules without leaving their big-data environment. By combining performance with full transparency, *Stickler* meets a growing need for trusted rule management in modern data platforms.

6.2. Future Work

While the rule engine developed in this thesis is complete and functional, demonstrating its capabilities effectively, there is still considerable scope for expansion and refinement. This section explores several potential avenues for future work that could build upon the current version of the *Stickler* rule engine.

The following areas are identified as key opportunities for future development:

1. **Evolution towards a Business Rule Management System (BRMS):** The current rule engine provides a robust backend for rule execution. A significant advancement would be its evolution into a more comprehensive BRMS. This could involve the development of a Graphical User Interface (GUI) or a web-based front-end. Such an interface would democratize rule management, allowing non-technical business users to define, modify, and manage rules without directly interacting with JSON configurations or code. An intermediate step towards this option could be the implementation of alternative rule input formats, such as CSV files structured as decision tables, which are often more intuitive for business analysts.

2. **Improved rule validation and edge case coverage:** While the current validation system addresses common issues, future efforts could focus on investigating and implementing more comprehensive validation checks. This would involve identifying and testing for a wider range of edge cases in rule definitions and their interactions, such as more complex type inference or potential ambiguities in rule logic. The goal would be to further increase the robustness of the engine and provide even more precise feedback to users during the rule definition phase.

3. **Implementation of an advanced rule optimization algorithm (such as Rete's):** To enhance performance, particularly with very large and complex rule sets, the integration of an advanced rule optimization algorithm, such as the Rete algorithm or its variants, could be investigated. The Rete algorithm

is designed to efficiently match a large collection of patterns against a large collection of objects, potentially reducing redundant computations by collapsing or reordering rules during the parsing or pre-processing stage, and it is already implemented in other BRE's, as it is the case for Drools'. This was not pursued in the current project due to its inherent complexity and the primary research focus on distributed data processing aspects with PySpark and cloud infrastructure, rather than on the theoretical foundations of rule engine algorithms themselves.

4. **Expansion of deployment options and their respective user guides:** Currently, the engine has been tested and documented for deployment on AWS Glue. Future work could involve extending deployment capabilities to other major cloud platforms in their respective equivalent, such as Google Cloud Platform (GCP) and Microsoft Azure, as well as to on-premise infrastructures. This would broaden the engine's applicability across different organizational environments. Furthermore, exploring deployment on container orchestration platforms like Amazon Elastic Kubernetes Service (EKS). Crucially, each new deployment option should be accompanied by detailed user guides and best-practice documentation to facilitate adoption.

5. **Exploration of real-time processing:** The present implementation primarily focuses on batch processing of large datasets, which is a common requirement for many business rule applications. However, extending the engine's capabilities to support real-time or near real-time stream processing would be a valuable addition. This would involve researching its integration and possible adaptation, with stream processing frameworks, such as Spark Streaming or Apache Flink, and adapting the rule execution logic to handle continuous data inflows, enabling use cases such as real-time fraud detection or dynamic customer personalization.

Bibliography

[1] Ahmed, N., Barczak, A. and Susnjak, T. A comprehensive performance analysis of Apache Hadoop and Apache Spark for large scale data sets using HiBench. *Journal of Big Data*, Vol. 7(110), 2020.

[2] Amazon Web Services. What is Distributed Computing? n.d. https://aws.amazon.com/what-is/distributed-computing/.

[3] Apache Software Foundation. Apache Beam Official Documentation. n.d.. https://beam.apache.org/documentation/.

[4] Apache Software Foundation. Apache Flink Official Documentation. n.d.. https://nightlies.apache.org/flink/flink-docs-lts/.

[5] Apache Software Foundation. MapReduce Tutorial (Hadoop). n.d.. https://hadoop.apache.org/docs/stable/hadoop-mapreduce-client/hadoop-mapreduce-client-core/MapReduceTutorial.html.

[6] Apache Software Foundation. Pyspark official documentation. n.d.. https://spark.apache.org/docs/latest/api/python/.

[7] Assessment, C. U. P. . Stickler. n.d. https://dictionary.cambridge.org/dictionary/english/stickler.

[8] AWS. Official Amazon EMR Documentation. n.d.. https://docs.aws.amazon.com/emr/.

[9] AWS. Official AWS Glue Documentation. n.d.. https://docs.aws.amazon.com/glue/.

[10] Badman, A. and Kosinski, M. What is Big Data? 2024. https://www.ibm.com/think/topics/big-data.

[11] Berlioz, C.-A. Business Rules Engine versus BRMS: The Repository. 2011. https://www.sparklinglogic.com/business-rules-engine-versus-brms-the-repository/.

[12] Bhati, M. Rule Engine for Healthcare: Transforming Medical Decision-Making. n.d. https://www.nected.ai/blog/rule-engine-for-healthcare.

[13] Black developers. Black - the uncompromising code formatter. n.d. https://black.readthedocs.io/en/stable/.

[14] Camunda. What is a Business Rules Engine: Benefits and Use Cases. 2024. https://camunda.com/blog/2024/07/the-business-process-rules-engine/.

[15] Chaos Genius. Apache Spark vs Flink—A Detailed Technical Comparison (2025). 2025. https://www.chaosgenius.io/blog/apache-spark-vs-flink.

[16] Chris Pitman, Justin Cohler, and Mauricio Navarro Miranda. spark-drools-example. 2015. https://github.com/cpitman/spark-drools-example, GitHub repository.

[17] Chuck Ballard, Daniel C. Boulanger, Henry Liu, Felipe Pio, and Andre N. Rodrigues. Implementing Big Data and Analytics on IBM Systems. 2014. https://www.redbooks.ibm.com/redbooks/pdfs/sg248359.pdf, IBM Redbooks publication.

[18] Databricks. What is PySpark? n.d. https://www.databricks.com/glossary/pyspark.

[19] DecisionRules. DecisionRules - Business Rules Engine. n.d. https://www.decisionrules.io.

[20] Deloitte Central Europe. Dynamic Pricing. n.d. https://www.deloitte.com/ce/en/services/consulting/services/dynamic-pricing.html.

[21] Fernando Doglio, Camunda. Understanding decision tables: A complete guide for beginners. *Camunda Platform 8 Blog*, 2024. https://camunda.com/blog/2024/12/understanding-decision-tables-a-complete-guide-for-beginners/.

[22] FICO. FICO Blaze Advisor. n.d. https://www.fico.com/en/products/fico-blaze-advisor.

[23] Forgy, C. L. Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem. *Artificial Intelligence*, Vol. 19(1), 17–37, 1982. https://www.csl.sri.com/users/mwfong/public_html/Technical/RETE%20Match%20Algorithm%20-%20Forgy%20OCR.pdf.

[24] Forrester Research. The Forrester New Wave™: Digital Decisioning Platforms, Q4 2018. 2018. https://www.scribd.com/document/402543656/The-Forrester-New-Wave-D-pdf.

[25] Foundation, A. S. Apache Spark Documentation. n.d. https://spark.apache.org/docs/latest/.

[26] Gad August on Medium. Why Apache Spark Outshines Hadoop in Big Data Processing. 2024. https://medium.com/@gadnwachukwu2/why-apache-spark-outshines-hadoop-in-big-data-processing-b47dbeabb049.

[27] Google. Google Python Style Guide. n.d. https://google.github.io/styleguide/pyguide.html, Google Open Source Project.

[28] Google Cloud. What is Big Data? n.d. https://cloud.google.com/learn/what-is-big-data.

[29] GoRules. BRE vs BRMS. n.d. https://docs.gorules.io/reference/bre-vs-brms.

[30] Harvard Business School Online. What is dynamic pricing? 2024. https://online.hbs.edu/blog/post/what-is-dynamic-pricing.

[31] Hewlett Packard Enterprise. What is Real-Time Processing. n.d. https://www.hpe.com/emea_europe/en/what-is/real-time-processing.html.

[32] IBM. ODM Business Rules with Apache Spark Batch operations. 2016. https://web.archive.org/web/20180831072224/https://developer.ibm.com/odm/docs/solutions/odm-and-analytics/odm-business-rules-with-apache-spark-batch-operations/, accessed through archive.org, snapshot from August 31st,2018.

[33] IBM. Decision tables. 2024. https://www.ibm.com/docs/en/odmoc?topic=tables-decision.

[34] IBM. Business Rule Parts and Structure. n.d.. https://www.ibm.com/docs/en/dbaoc?topic=rules-business-rule-parts-structure.

[35] IBM. IBM Operational Decision Manager. n.d.. https://www.ibm.com/products/operational-decision-manager.

[36] InRule. If/Then Rule. 2025. https://support.inrule.com/hc/en-us/articles/4406952214541-If-Then-Rule.

[37] Kaggle. Kaggle - Your home for Data Science. n.d. https://www.kaggle.com/.

[38] Kakas, A. Building Dynamic Pricing for Fortune 500 Specialty Retailer - Case Study. 2023. https://www.revologyanalytics.com/articles-insights/dynamic-pricing-fortune500-retailer-case.

[39] Marcu, O.-C., Costan, A., Antoniu, G. and Pérez-Hernández, M. S. Spark versus Flink: Understanding Performance in Big Data Analytics Frameworks. 2016. https://inria.hal.science/hal-01347638v1/document.

[40] Martin Fowler. Given-when-then. 2013. https://martinfowler.com/bliki/GivenWhenThen.html.

[41] Mesbahi, N., Zoubeidi, M., Merizig, A. and Kazar, O. An agent-based approach for extracting business association rules from centralized databases systems. *Journal of Digital Information Management*, Vol. 17(5), 270–288, 2019.

[42] NECTED AI. How are Rules Engines Used in the Banking Industry? n.d.. https://www.nected.ai/blog/rules-engines-used-in-banking-industry.

[43] NECTED AI. What Is a Business Rule Engine? n.d.. https://www.nected.ai/blog/top-10-business-rules-engine.

[44] ŁUKASZ NIEDOŚPIAŁ. Business Rules Engine Technology In Cargo Logistic. 2024. https://www.higson.io/blog/business-rules-engine-technology-in-cargo-logistic.

[45] NOWAK, M. Business Rules Engines vs. Traditional Rule Management Systems. 2024. https://www.higson.io/blog/business-rules-engines-vs-traditional-rule-management-systems.

[46] OBJECT MANAGEMENT GROUP. Decision Model and Notation (DMN). n.d. https://www.omg.org/dmn/.

[47] OPENL TABLETS. OpenL Tablets - Easy Business Rules. n.d. https://openl-tablets.org.

[48] PEGA. Decision Management Overview. n.d. https://academy.pega.com/topic/decision-management-overview/v1.

[49] PIERRE FEILLET (@PIERREFEILLET ON GITHUB, PRODUCT ARCHITECT, IBM OPERATION DECISION MANAGER) AND LAURENT GRATEAU (@LGRATEAU). decisions-on-spark. n.d. https://github.com/DecisionsDev/decisions-on-spark/blob/master/README.md, GitHub repository. Licensed under Apache License 2.0.

[50] PYDANTIC DEVELOPERS. Pydantic Documentation. n.d. https://docs.pydantic.dev.

[51] PYLINT DEVELOPERS. Pylint - Python static code analysis tool. n.d. https://pylint.pycqa.org/en/latest/.

[52] PYTEST DEVELOPERS. Pytest - Python testing framework. n.d. https://docs.pytest.org/en/stable/.

[53] RAMASAHAYAM, R. R. Automate Decisions using Apache Spark. 2021. https://www.red-gate.com/simple-talk/blogs/automate-decisions-using-apache-spark/.

[54] RED HAT. Drools - Business Rules Management System. n.d. https://www.drools.org.

[55] REFACTORING.GURU. Abstract Factory Design Pattern. n.d.. https://refactoring.guru/design-patterns/abstract-factory.

[56] REFACTORING.GURU. Chain of Responsability Design Pattern. n.d.. https://refactoring.guru/design-patterns/chain-of-responsibility.

[57] Refactoring.Guru. Facade Design Pattern. n.d.. https://refactoring.guru/design-patterns/facade.

[58] RisingWave. Apache Beam vs Apache Spark: Which Data Processing Framework is Right for You? 2023. https://risingwave.com/blog/apache-beam-vs-apache-spark-which-data-processing-framework-is-right-for-you, RisingWave Blog.

[59] Sapiens. It's Official: Business Rules Engines are Out, Decision Automation is In. n.d. https://sapiens.com/blog/its-official-business-rules-engines-are-out-decision-automation-is-in/.

[60] Shashwat Pandey. Apache Spark vs Apache Beam: What to Choose When? 2021. https://shashwat-pandey.medium.com/apache-spark-vs-apache-beam-what-to-choose-when-94f938d0317f, Medium article.

[61] Solvexia. Business Rules Engines. 2024. https://www.solvexia.com/glossary/business-rules-engines.

[62] Tentoro AI. Business Rules Engine: Automating Processes and Decisions. n.d. https://tentoro.ai/blog/business-rules-engine-automating-processes-and-decisions.

[63] The Apache Software Foundation. Parallel Execution in Apache Flink. n.d. https://nightlies.apache.org/flink/flink-docs-release-1.20/docs/dev/datastream/execution/parallel/.

[64] Timothy Crosley. isort - Python library for sorting imports. n.d. https://pycqa.github.io/isort/.

[65] TutorialsPoint. Hadoop - MapReduce. n.d. https://www.tutorialspoint.com/hadoop/hadoop_mapreduce.htm.

[66] Wikipedia contributors. Backward Chaining. 2013. https://en.wikipedia.org/wiki/Backward_chaining.

[67] Wikipedia contributors. Forward Chaining. 2013. https://en.wikipedia.org/wiki/Forward_chaining.

Appendix A

JSON input example

Listing A.1: JSON input for Performance Evaluation

```
{
  "rules":[
    {
      "rule_name":"SummerEconomyDiscount",
      "execution_type":"ACCUMULATIVE",
      "cascade_group":1,
      "conditions":[
        {
          "expression":"month(flightDate) IN (6,7,8)"
        },
        {
          "expression":"isBasicEconomy = true OR segmentsCabinCode
            LIKE '%coach%'"
        },
        {
          "expression":"seatsRemaining > 5"
        }
      ],
      "actions":[
        {
          "output_col_name":"totalFare",
          "operation":"totalFare * 0.90"
        }
      ]
    },
    {
      "rule_name":"LastMinuteDeal",
      "execution_type":"CASCADE",
      "cascade_group":1,
      "conditions":[
        {
          "expression":"datediff(flightDate, searchDate) <= 3"
        }
      ],
      "actions":[
        {
          "output_col_name":"totalFare",
```

```
37             "operation":"totalFare * 0.85"
38           }
39         ]
40       },
41       {
42         "rule_name":"AdvanceBookingPremium",
43         "execution_type":"CASCADE",
44         "cascade_group":1,
45         "conditions":[
46           {
47             "expression":"datediff(flightDate, searchDate) >= 60"
48           }
49         ],
50         "actions":[
51           {
52             "output_col_name":"totalFare",
53             "operation":"totalFare * 1.10"
54           }
55         ]
56       },
57       {
58         "rule_name":"RefundableTicketSurcharge",
59         "execution_type":"ACCUMULATIVE",
60         "cascade_group":0,
61         "conditions":[
62           {
63             "expression":"isRefundable = true"
64           }
65         ],
66         "actions":[
67           {
68             "output_col_name":"totalFare",
69             "operation":"totalFare + 20"
70           }
71         ]
72       },
73       {
74         "rule_name":"AirportHubDiscount",
75         "execution_type":"ACCUMULATIVE",
76         "cascade_group":0,
77         "conditions":[
78           {
79             "expression":"startingAirport IN ('ATL','DFW','ORD','LAX
                 ')"
80           }
81         ],
82         "actions":[
83           {
84             "output_col_name":"totalFare",
85             "operation":"totalFare * 0.97"
86           }
87         ]
88       },
89       {
90         "rule_name":"HighDemandSurcharge",
91         "execution_type":"ACCUMULATIVE",
```

```
92          "cascade_group":2,
93          "conditions":[
94            {
95              "expression":"seatsRemaining < 5"
96            }
97          ],
98          "actions":[
99            {
100             "output_col_name":"totalFare",
101             "operation":"totalFare * 1.15"
102           }
103         ]
104       },
105       {
106         "rule_name":"OffPeakDayDiscount",
107         "execution_type":"CASCADE",
108         "cascade_group":2,
109         "conditions":[
110           {
111             "expression":"dayofweek(flightDate) IN (3,4,5)"
112           }
113         ],
114         "actions":[
115           {
116             "output_col_name":"totalFare",
117             "operation":"totalFare * 0.92"
118           }
119         ]
120       },
121       {
122         "rule_name":"LongHaulBusinessDiscount",
123         "execution_type":"ACCUMULATIVE",
124         "cascade_group":0,
125         "conditions":[
126           {
127             "expression":"segmentsCabinCode LIKE '%business%'"
128           },
129           {
130             "expression":"totalTravelDistance > 2500"
131           }
132         ],
133         "actions":[
134           {
135             "output_col_name":"totalFare",
136             "operation":"totalFare - 50"
137           }
138         ]
139       },
140       {
141         "rule_name":"ExtendedLayoverDiscount",
142         "execution_type":"ACCUMULATIVE",
143         "cascade_group":3,
144         "conditions":[
145           {
146             "expression":"size(split(
                  segmentsDepartureTimeEpochSeconds, '||')) > 1"
```

```
147         },
148         {
149           "expression":"max_layover_hours(
                segmentsDepartureTimeEpochSeconds,
                segmentsArrivalTimeEpochSeconds) > 1.5"
150         },
151         {
152           "expression":"seatsRemaining > 5"
153         }
154       ],
155       "actions":[
156         {
157           "output_col_name":"totalFare",
158           "operation":"totalFare * 0.93"
159         },
160         {
161           "output_col_name":"layover_hours",
162           "operation":"round((cast(split(
                segmentsDepartureTimeEpochSeconds,'\\\\|\\\\|')[1] AS
                double)-cast(split(segmentsArrivalTimeEpochSeconds,
                '\\\\|\\\\|')[0] AS double))/3600,1)",
163           "otherwise":"0"
164         }
165       ]
166     },
167     {
168       "rule_name":"RedEyeSeatSale",
169       "execution_type":"CASCADE",
170       "cascade_group":[
171         1,
172         3
173       ],
174       "conditions":[
175         {
176           "expression":"isBasicEconomy = True OR segmentsCabinCode
                LIKE '%coach%'"
177         },
178         {
179           "expression":"totalFare > 50"
180         }
181       ],
182       "actions":[
183         {
184           "output_col_name":"totalFare",
185           "operation":"apply_redeye_discount(
                segmentsDepartureTimeRaw, totalFare)"
186         }
187       ]
188     }
189   ]
190 }
```

Appendix B

AWS Glue Script code

Listing B.1: AWS Glue Script code

```
import sys
import json
from awsglue.transforms import *
from awsglue.utils import getResolvedOptions
from awsglue.context import GlueContext
from awsglue.job import Job
from pyspark.context import SparkContext
from pyspark.sql.types import DoubleType
from pyspark.sql.functions import udf
from stickler.engine import RuleEngine, RulesConfig

args = getResolvedOptions(sys.argv, ["JOB_NAME"])

sc = SparkContext.getOrCreate()
glueContext = GlueContext(sc)
spark = glueContext.spark_session

job = Job(glueContext)
job.init(args["JOB_NAME"], args)


df_input = (spark
    .read
    .option("header", "true")
    .option("inferSchema", "true")
    .csv("s3://test-bucket-stickler/flights-test/itineraries.csv")
)


rules_json = """JSON file content can be found in Appendix A"""
rules_json = json.loads(rules_json)
rules_config = RulesConfig(**rules_json)

# Define UDFs
def apply_redeye_discount(segments_raw: str, fare: float) -> float:
    # Ensure fare is a float
    fare_val = float(fare)
```

```
38     # Check each leg's departure hour by slicing characters 11-13
          from ISO format date
39     for part in segments_raw.split("||"):
40         try:
41             hour = int(part[11:13])
42         except (ValueError, IndexError):
43             continue
44         if hour < 6:
45             return fare_val * 0.88
46     return fare_val
47 
48 def max_layover_hours(dep_str, arr_str):
49     deps = [int(x) for x in dep_str.split("||")]
50     arrs = [int(x) for x in arr_str.split("||")]
51     layovers = []
52     for i in range(len(arrs)-1):
53         layovers.append((deps[i+1] - arrs[i]) / 3600.0)
54     return max(layovers or [0.0])
55 
56 # Register the UDF with Spark
57 apply_redeye_discount_udf = udf(apply_redeye_discount, DoubleType()
     )
58 max_layover_udf = udf(max_layover_hours, DoubleType())
59 
60 udfs = {
61     "apply_redeye_discount": apply_redeye_discount_udf,
62     "max_layover_hours": max_layover_udf
63 }
64 
65 # Start the engine
66 engine = RuleEngine(rules_config, udfs)
67 output_df, history_df = engine.apply_rules(df_input)
68 
69 # Write the output
70 output_df.write.mode("overwrite") \
71     .parquet("s3://test-bucket-stickler/flights-test/output/
          output_df/")
72 history_df.write.mode("overwrite") \
73     .parquet("s3://test-bucket-stickler/flights-test/output/
          history_df/")
74 
75 job.commit()
```

Appendix C

User Guide

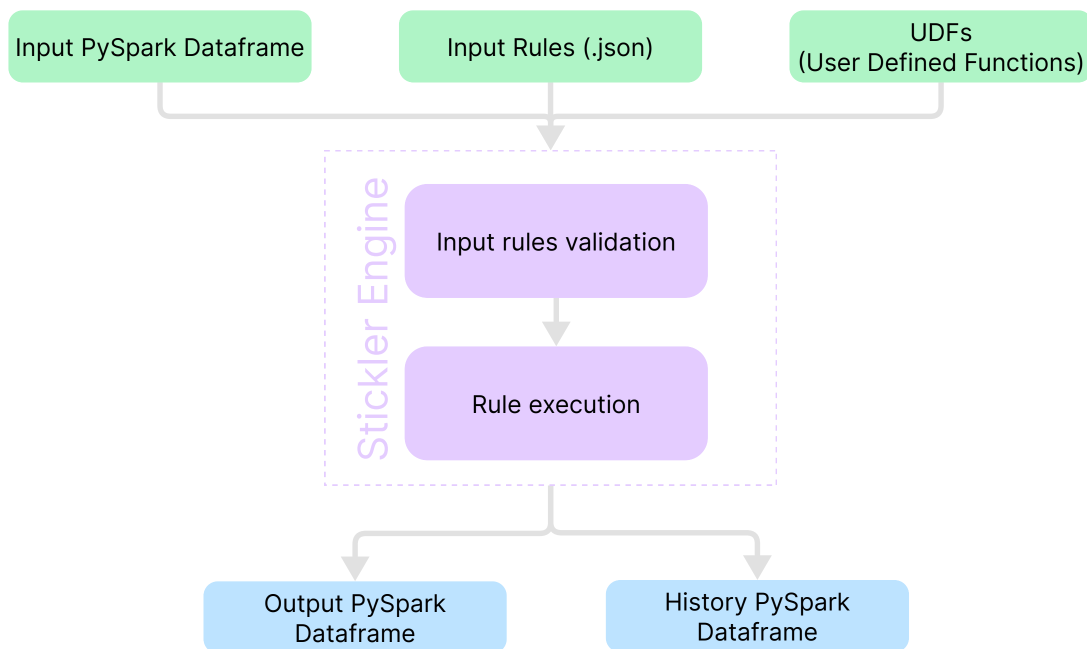

Figure C.1: Overview of Stickler's operational workflow.

The rule engine takes in an input PySpark DataFrame, a JSON file defining rules, and optional User-Defined Functions (UDFs); the engine then validates these inputs and executes the rules, subsequently producing two output PySpark DataFrames: one with the transformed data and another detailing the execution history.

Installation

Apache Spark and Python are required to run Stickler. You can then easily install the rule engine using pip:

```
pip install stickler
```

Usage

Importing Stickler

Once installed, import the engine in your Python script (`main.py` or similar):

```
from stickler import engine
```

Initializing the Engine

Load the rules from a configuration JSON file and initialize the RuleEngine:

```
import json
from pyspark.sql import SparkSession

# Initialize Spark Session
spark = SparkSession.builder.appName("SticklerApp").getOrCreate()

# Load rules from a JSON file
with open("rules.json") as f:
    rules_data = json.load(f)

# Parse JSON into RulesConfig object
rules_configuration = engine.RulesConfig(**rules_data)

# Instantiate the engine
rule_engine_instance = engine.RuleEngine(rules_configuration)
```

Using User-Defined Functions (UDFs)

Optionally, Stickler supports PySpark UDFs in rule expressions. Define your UDFs and pass them to the engine as a dictionary:

```
from pyspark.sql.functions import udf

# Example UDF definitions
def custom_upper(value: str) -> str:
    return value.upper()

def get_string_length(value: str) -> int:
    return len(value)

# Register UDFs with Spark
udf_to_upper = udf(custom_upper, StringType())
udf_string_length = udf(get_string_length, IntegerType())
udfs_dictionary = {
    "to_upper": udf_to_upper,
    "string_length": udf_string_length
}

# Initialize engine with UDFs
rule_engine_instance_with_udfs = engine.RuleEngine(
   rules_configuration, udfs=udfs_dictionary)
```

Executing the Engine (Apply Rules)

Assuming `input_df` is the input PySpark DataFrame:

```
output_df, history_df = rule_engine_instance.apply_rules(input_df)
```

JSON Decision Model (JDM) for Rule Definition

Overview

Stickler uses a JSON-based format for defining business rules. This allows for easy configuration and management of rules in a structured, human-readable way. The JDM is a single JSON object containing a primary key, `"rules"`, which holds an array of rule definition objects.

Rules are evaluated **sequentially** in the order they were defined, from **top to bottom**.

Structure

Each rule object within the `"rules"` array has the following structure:

```
{
    "rules": [
        {
            "rule_name": "rule_name1",
            "execution_type": "cascade"/"accumulative",
            "cascade_group": [0, 1, 2] / 0 (optional),
            "conditions": [
                {
                    "expression": "spark_sql_expression"
                }
            ],
            "actions": [
                {
                    "output_col_name": "column_name",
                    "operation": "spark_sql_expression",
                    "otherwise": "spark_sql_expression" (optional)
                }
            ]
        },
        ...
    ]
}
```

- **rule_name** (string): A unique and descriptive name for the rule.
- **execution_type** (string): Determines how the rule interacts with others.
    - `"accumulative"`: The rule is applied independently to every row satisfying its conditions. Default if unspecified.

- "cascade": Executes only if no other rule in the same cascade_group has been applied to that row.

- **cascade_group** (integer or list of integers, optional): Identifies group(s) to manage rule precedence. Defaults to 0.
- **conditions** (array of objects): All must evaluate to true for actions to trigger.
  - **expression** (string): A Spark SQL boolean expression, e.g., "original_price > 100".
- **actions** (array of objects): Defines column modifications.
  - **output_col_name** (string): The column to update or create.
  - **operation** (string): Spark SQL expression applied if conditions are met.
  - **otherwise** (string, optional): Alternative operation if conditions fail.

Rule Execution Types

Each rule can be executed in one of two modes:

- **Accumulative:** Applies to every row that meets its conditions, independently of other rules.
- **Cascade:** Uses cascade_group to enforce precedence. A rule is applied only if no other rule in the same group has already been applied to that row.

Rule Validation

Before executing any rules, Stickler performs validation on the rule definitions provided in the JSON file to detect and prevent errors early. The validation process includes the following checks:

- **Syntax Errors in JSON:** Ensures that the rule definitions have the correct structure and contain all required fields, confirming the JSON is valid.
- **Name Validation:** Rule names must be non-empty, unique, and must not conflict with internal Stickler naming conventions (e.g., those used in the history DataFrame, typically named RuleName_OutputColumnName). This step flags duplicate or missing rule names.
- **Reference Validation:** Verifies that all columns referenced in the rule's conditions or actions exist—either in the input DataFrame or as outputs from previous rules. An error is raised if any referenced column is missing.
- **Expression Validation:** Detects common issues such as invalid operators or syntax errors in Spark SQL expressions used within rules.

History Tracking

Stickler maintains a history DataFrame that tracks the behavior and impact of each rule during execution. This DataFrame includes:

- All original columns from the input DataFrame.
- A boolean column for each rule, named after the rule itself, indicating whether that rule was applied to each row (`True` if applied, `False` otherwise).
- Additional columns for each action that modifies or creates data. These columns follow the naming format `RuleName_OutputColumnName` and contain the value assigned to the output column by the corresponding rule.
- For example, the column `discountPrices_Price` would represent the output of the rule `discountPrices` applied to the `Price` column.

If a rule's conditions are met but it is blocked from execution due to cascade logic (i.e., another rule in the same group was already applied to the row), it will be marked as not executed (`False`) in its corresponding history column.