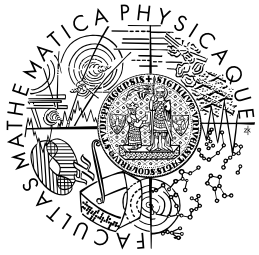

FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

BACHELOR THESIS

Zdeněk Tomis

Streamlining Usability of Enterprise Data Quality Management Tools for Data Engineers

Department of Distributed and Dependable Systems

Supervisor of the bachelor thesis: doc. Ing. Lubomír Bulej, Ph.D.

Study programme: Computer science

Prague 2024

I declare that I carried out this bachelor thesis on my own, and only with the cited sources, literature and other professional sources. I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In ..... date ....

Author's signature

I would like to dedicate this thesis to my family and friends, whose support has been and continues to be priceless. Special thanks go to my supervisor, doc. Ing. Lubomír Bulej, Ph.D., for his guidance, flexibility, and understanding. I also want to express my gratitude to colleagues in Ataccama for their support, namely to Lukáš Kolek and Ondřej Chrastina.

Title: Streamlining Usability of Enterprise Data Quality Management Tools for Data Engineers

Author: Zdeněk Tomis

Department: Department of Distributed and Dependable Systems

Supervisor: doc. Ing. Lubomír Bulej, Ph.D., Department of Distributed and Dependable Systems

Abstract: In the realm of data quality management, integrating robust data quality rules into automated workflows and data pipelines is essential for maintaining data integrity. This thesis addresses the gap in programmatic accessibility of Ataccama ONE's data quality tools, which primarily leverage the proprietary Ataccama Expression Language. By reimplementing this language in Python, the project enhances its usability for data engineers who seek to consume these tools programmatically. The focus is on enabling data engineers to execute Ataccama's rules directly within Python. The viability of this implementation is tested through performance comparisons with similar solutions.

Keywords: data quality management, data engineering, performance evaluation

Název práce: Zjednodušení použitelnosti nástrojů pro správu kvality dat pro datové inženýry

Autor: Zdeněk Tomis

Katedra: Katedra distribuovaných a spolehlivých systémů

Vedoucí bakalářské práce: doc. Ing. Lubomír Bulej, Ph.D., Katedra distribuovaných a spolehlivých systémů

Abstrakt: V oblasti data quality managementu je pro zachování integrity dat zásadní integrovat pravidla kvality dat do automatizovaných workflows a datových pipelines. Tato práce se zabývá mezerou v programové dostupnosti nástrojů pro kvalitu dat společnosti Ataccama ONE, které využívají především proprietární jazyk Ataccama Expression Language. Reimplementací tohoto jazyka v jazyce Python projekt zvyšuje jeho použitelnost pro datové inženýry, kteří potřebují tyto nástroje využít programmaticky v různým prostředích. Důraz je kladen na to, aby datoví inženýři mohli provádět a spravovat pravidla Ataccama přímo v jazyce Python s ohledem na jednoduchost užití a minimální nároky. Užitelnost této implementace je otestována prostřednictvím porovnání výkonu s podobnými řešeními.

Klíčová slova: data quality management, data engineering, performance evaluation

Contents

Introduction

DQM (Data Quality Management) refers to the processes, technologies, and practices used to maintain high quality in data through its lifecycle. It encompasses the acquisition, implementation, and control of data accuracy, completeness, reliability, and relevance in enterprise systems. DQM ensures that data remains accurate, consistent, and accessible across all platforms and applications within an organization.

In the age of big data and advanced analytics, DQM is not just a luxury—it's an imperative. It is critical for modern enterprises for many reasons, including the following:

- Informed Decision-Making

  High-quality data is pivotal for accuracy in decision-making. Decisions based on inaccurate or incomplete data can lead to significant financial losses and strategic missteps.

- Regulatory Compliance

  Many industries are subject to regulations that mandate the integrity and confidentiality of data. For example, the GDPR in Europe and HIPAA in the United States impose strict guidelines on data privacy and the quality of information that is stored and processed. DQM helps organizations comply with these regulations and avoid hefty penalties by ensuring data is managed correctly throughout its lifecycle.

- Enhanced Customer Satisfaction

  Data quality directly impacts customer experience. Accurate customer data helps businesses better understand their clients and better tailor their interactions, improving customer satisfaction and loyalty.

- Operational Efficiency

  High-quality data reduces errors and the need for rework. For instance, accurate inventory data helps manage stock levels efficiently, avoiding overstocking or stockouts. By automating data cleansing and enrichment, organizations can streamline workflows and allow employees to focus on higher-value activities rather than correcting data errors.

- Risk Mitigation

  Poor data quality is a significant risk — it can skew analysis, leading to misguided strategies that may harm the business. DQM practices identify and correct discrepancies in data before they propagate through the enterprise, thereby mitigating risks associated with data handling and storage.

The need for programmatic access

In the context of DQM, most commonly, the center point of the focus is on the persona of the so-called 'data steward.' [1] This is because data stewards are

crucial for any data quality initiative. Data stewards must be involved in setting up data quality processes to ensure success. This includes tasks such as defining data quality rules, profiling, and addressing data quality issues.

In the world of DQM, there arises a need to consume data quality tools programmatically. This is where the persona of a data engineer comes into play. There are more reasons for this, in simple terms, the necessity is a corollary to the need to integrate data quality tools into data pipelines, which is what data engineers are responsible for.

It follows that any solution aimed at pipeline integration should be designed to accommodate data engineers as they will be its users. In the following sections, we will discuss the needs of data engineers in the context of DQM tools.

Data Quality Management Tools at Ataccama

Ataccama stands out in the field of DQM (Data Quality Management) due to its comprehensive suite of tools designed to enhance data quality across multiple dimensions. Ataccama ONE, the flagship product, is renowned for its robust data profiling, quality rule enforcement, monitoring, and issue resolution capabilities. This platform serves as a critical tool for enterprises looking to maintain high standards of data integrity.

However, one of the primary limitations of Ataccama ONE is its primary reliance on a web interface, which poses challenges for seamless integration into automated data pipelines typically managed by data engineers. This limitation highlights the need for programmatic access to its powerful DQM features to fully leverage its capabilities in more dynamic, code-driven environments.

Data Quality Rules

Ataccama ONE's capacity to define complex data quality rules using its expression language is one of its most potent features. These expressions allow for precise specifications of data quality requirements that can dynamically adapt to varying data sets and conditions. This expression language supports an extensive range of functions and operators, enabling detailed data validation and cleansing processes that are crucial for reliable data analytics.

Integrating these expression-based rules into typical data engineering workflows could significantly streamline processes involved in data validation and correction, providing a more robust framework for ensuring data quality at scale. Enabling programmatic access to these rules would allow data engineers to automate and integrate data quality checks directly into data ingestion and processing pipelines, thereby enhancing efficiency and reducing the likelihood of errors.

Personal Motivation and Practical Applications

The focus on Ataccama in this thesis is influenced by practical and personal considerations. As a developer relations employee at Ataccama, I have unique access to the platform and a vested interest in exploring its capabilities deeply. This position provides a unique opportunity to bridge the gap between Ataccama's

current offerings and the needs of data engineers who require more flexibility and programmatic control over their data quality tools.

By leveraging Ataccama's existing robust framework and extending its accessibility to data engineers through programmatic interfaces, this project aligns with industry best practices while pushing the boundaries of traditional data quality management. The goal is to transform Ataccama from a primarily UI-driven tool into a versatile backend service that can power complex data pipelines, making it an even more valuable asset in the data engineer's toolkit.

Objective of the thesis

This thesis aims to significantly enhance the accessibility of Ataccama's data quality management tools for data engineers by developing a programmatic interface that allows for direct execution and integration of Ataccama's robust expression language within automated data workflows. The expected outcome is to provide data engineers with the tools they need to enforce data quality seamlessly within their pipelines. By doing so, this work not only extends the usability of Ataccama ONE within modern data-driven environments but also contributes to the broader field of data quality management by bridging the gap between advanced data quality rules and operational data engineering practices.

1 Analysis

There are many benefits to having DQM (Data Quality Management) processes in place. In fact, in today's data-driven environments, the assurance of data quality is not just a preference but a critical necessity. Organizations rely on accurate, timely, and reliable data to make informed decisions, drive strategies, and optimize operations. As such, the field of DQM has evolved to address these needs through sophisticated tools and methodologies. However, the effective implementation of these tools requires a deep understanding of both the tools themselves and the roles of those who interact with them.

This chapter delves into the analysis of the current landscape of DQM, focusing particularly on the need for programmatic access to these tools. This need stems from the growing requirement to seamlessly integrate data quality solutions into existing data pipelines, a task that typically falls within the purview of data engineers. As we explore this topic, we will discuss the challenges faced by data engineers.

1.1 Similar Solutions

This section compares Ataccama ONE to other DQM tools that are designed for programmatic access, highlighting the relative strengths and limitations of each solution in terms of features, technology stack, and integration capabilities.

1.1.1 Soda Core

Soda Core [2] is a robust open-source tool tailored to integrate data quality checks directly into data pipelines. However, its feature set primarily focuses on:

- Data Monitoring and Alerting

  Automatically detecting and alerting on anomalies in data as it flows through pipelines.

- Customizable Quality Checks

  While flexible, these are generally more basic compared to the depth provided by comprehensive DQM platforms.

- Python Integration

  Strong integration with Python-based data ecosystems, suitable for teams relying heavily on Python for data processing.

1.1.2 Great Expectations

Great Expectations [3] offers a framework for setting up complex data validation and documentation, which is crucial for maintaining high data quality standards. Its key features include:

- Validation Framework

  Extensive support for defining expectations about data, which can be automatically validated against data batches.

- Data Docs

  Automatically generated documentation that helps keep teams aligned on data quality standards.

- Integration

  While it offers broad integration capabilities, it requires significant setup and maintenance compared to more out-of-the-box solutions.

1.1.3 Comparison with Ataccama ONE

Ataccama ONE offers a more extensive suite of features than either Soda Core or Great Expectations. This includes advanced functionalities like Master Data Management, Data Cataloging, and enriched Metadata Management, which are not typically found in the aforementioned open-source tools. However, also the data quality features are more comprehensive, more advanced functions are included out of the box, including but not limited to: reference data checks, complex string manipulation, and set operations.

Despite its robust feature set, Ataccama's main limitation lies in its less flexible integration with Python-based data pipelines, which is where Soda Core and Great Expectations excel due to their native Python support. This limitation is a significant drawback for data engineers who rely heavily on Python for data processing and pipeline orchestration and for this reason this thesis will focus on enhancing Ataccama's integration capabilities with Python-based data pipelines.

1.2 The persona of Data engineer and their needs

As previously noted, data engineers play a crucial role in integrating data quality tools into data pipelines. It is essential to design applications that meet the specific needs and requirements of its primary users. Therefore, any solution intended for pipeline integration must be carefully tailored to accommodate the unique needs of data engineers. This ensures that the tools not only fit seamlessly into their workflows but also enhance their efficiency and effectiveness in managing data quality across systems.

1.2.1 Used technologies

Data engineers, like any others, are working professionals. Many of them are used to working with some particular tools and technologies. It is vital to take into account the tools and technologies that data engineers are familiar with when designing a solution for them.

Although among the many data engineers, there are differences in the tools and technologies they use, there are some common tools and technologies that are used by the majority of them in today's world. It is important to realize that we

should not only consider an ideal data engineer, a skilled senior, but also consider that there are many junior data engineers. Also, it is important to consider the cognitive load of new tools and technologies on data engineers. Not only should the tools and technologies be easy to learn and use, but also they should be based on familiar concepts and technologies. This way, the data engineers can focus on their work and not on learning new tools and technologies.

The most commonly used tools and technologies by data engineers are Python, Java, Scala, and SQL. These tools and technologies are used for the development of data pipelines and the integration of data quality tools into these pipelines. For any set of data engineers, the intersection of their knowledge bases will include Python more often than anything else.

1.2.2 Ease of use

In any software, a balance between ease of use and functionality complemented by the correctness of conceptual abstraction needs to be found. As the API surface of this solution is not going to be large, not a lot of decisions will have to be made on this front. However, it is important to keep in mind that the users of the solution are data engineers, some of whom work in a consultant background. Many of them are not used to advanced language features, and abstractions have limited experience with software engineering. For this reason, it is important to keep the solution simple and easy to use, avoiding any complex constructs and patterns.

Furthermore, it should be taken into account that the library can be used outside of IDE and similar environments where code suggestion and documentation might not be available. For example, when writing integrations in the platforms discussed below such as Azure Data Factory and Data Bricks, the user will not have access to the documentation or code suggestions. For this reason, the API should be designed to be self-explanatory, and should be designed to be easy to use without the need for extensive documentation.

1.2.3 Data security

When designing data quality solutions for integration into existing data pipelines, especially those that involve interfacing with external applications or servers, security is a paramount concern. This is particularly critical when the data involved is sensitive, as is often the case in industries such as finance, healthcare, and government.

Sending data over the internet to a third-party service can be a security risk. Data security is a major concern for data engineers, and it is important to take into account the security requirements of data engineers when designing a solution for them.

When a data quality integration in a pipeline needs to access the server - an application running somewhere else - the network of the server has to be accessible. This can be a security risk as every new network endpoint provides an additional entry point for attackers. When applications within a private network start communicating with external entities, these points of interaction need to be secured, adding complexity and potential for oversight. If the networked application has

vulnerabilities, such as insufficient authentication, flawed authorization practices, or software bugs, it could be exploited by attackers to gain unauthorized access. This could lead to data breaches, data loss, or malicious data manipulation.

Additionally, data transmitted over networks can be intercepted, viewed, or altered by unauthorized parties if not adequately protected. This risk is particularly severe if data is transmitted over unsecured or improperly secured connections, such as those not using TLS/SSL protocols.

1.2.4 Ease of configuration

The need to access a running instance of a DQM application in order to run data quality tooling comes with added complexities.

First, the application needs to be configured and running. This is fine for an environment where such an application is already in use. Yet, still, it is an added complexity as part of the process is running somewhere else, so it can be more difficult to debug, monitor, and maintain.

Second, the pipeline needs to access the application over a network. This means that the application needs to be exposed to the network, which can not only be a security risk but also provide further complexities in terms of network configuration. In case of the server being accessible only on a private network, the application needs to be exposed to the network, which can, in some cases, be even out of question and make the integration impossible, or it can be an obstacle on the way to successful integration.

1.3 Data pipelines and requirements for the integration of data quality tools

Data pipelines are a crucial part of any data engineering project. Data pipelines are used to move data from one place to another and to transform data from one format to another.

Many of the use cases for integrating data quality tools into data pipelines include the requirement to integrate with existing data pipeline or solutions. The data quality tools should support integration into commonly used data pipelines. It also follows that forcing a new data pipeline or ETL (Extract-Transform-Load) solution is not a valid requirement.

For example, in Ataccama, the application is intended to be connected to all the data sources and data targets using its custom connectors. To access the Ataccama engine and run any sort of evaluation of data quality rules, the data must be loaded into Ataccama ONE using a connection setup within the application, or Ataccama can pushdown processing directly into databases or the data needs to be sent into a service set up from within the application. Either way, there is no straightforward way to process the data directly in the data stream because Ataccama current solution is oriented more toward table processing. In summary, all of these approaches present a challenge for integrating Ataccama into existing data pipelines.

1.3.1 Pipelines in commonly used data platforms

Modern data ecosystems are diverse, with organizations leveraging a variety of data storage solutions and computing environments to manage and analyze data. Here's how Python integration plays a critical role across commonly used platforms:

- Snowflake

  Snowflake[4] supports multiple programming languages, including Java and .NET, but Python remains a popular choice due to its extensive library support and community.

  Python is well-supported in Snowflake through connectors like Snowflake Connector for Python, which allows executing SQL statements and performing data manipulations directly from Python scripts.

- AWS Glue

  AWS Glue[5] supports Python and Scala. Python, being one of the main languages supported by AWS Glue, benefits from seamless integration with other AWS services.

  Python scripts in AWS Glue can perform ETL tasks effectively, which can be developed and debugged directly in Python using Glue's development endpoints.

- Azure Data Factory

  Azure Data Factory[6] supports custom activities in various languages, but Python's use in Azure functions for custom processing activities is notable due to its simplicity and effectiveness.

  Python in ADF can be used to orchestrate complex data workflows, invoking Python-based processes as part of the data integration pipelines.

- Databricks

  Databricks[7] offers a unified analytics platform that supports Python, Scala, SQL, and R. Python's integration, particularly with PySpark for big data processing, makes it a primary choice for many developers.

  Python is extensively used in Databricks notebooks for data exploration, visualization, and machine learning, highlighting its versatility and ease of use.

Given the need to operate within commonly used compute platforms such as the above-mentioned, it is imperative to consider the compatibility of programming languages supported by these environments. Each platform offers support for various technologies; however, Python stands out due to its universal acceptance and extensive integration capabilities across these systems. Whether it is executing complex data manipulation tasks in Snowflake, orchestrating ETL processes in AWS Glue, running custom activities in Azure Data Factory, or performing data analysis and machine learning in Databricks, Python is consistently supported.

Therefore, focusing on Python to implement data quality rules not only aligns with the operational capabilities of these platforms but also ensures that our

solutions are versatile and adaptable across different technological ecosystems. This strategic choice maximizes the utility and reach of our DQM tools, making them accessible and functional within the predominant data processing frameworks employed by contemporary organizations.

1.4 Enhancing Ataccama's Integration Capabilities

While Ataccama offers a rich set of data quality management features, one critical area where enhancement is needed is in its programmatic accessibility. This thesis sets out to address this limitation by focusing on the development of methodologies that will enable better integration into automated data environments, particularly through the reimagination of Ataccama's data quality expression language.

The core of this enhancement strategy involves reimplementing Ataccama's expression language in a way that maintains full compatibility with the original system. The intention is not to build entirely new features but to translate the existing capabilities into forms that are more accessible for programmatic use. This effort requires careful consideration to ensure that all functionalities remain consistent with Ataccama's established methods, thus preserving the integrity and reliability of the platform while extending its usability.

This focus on reimplementing the expression language aims to facilitate the direct application of Ataccama's powerful data quality rules within more diverse programming environments. By doing so, the project seeks to bridge the gap between Ataccama's robust data management tools and the practical, operational needs of modern data pipelines, making it more adaptable for data engineers who need to incorporate sophisticated data quality checks directly into their workflows. The outcome will be a more flexible tool that fits seamlessly into existing data infrastructures, enhancing Ataccama's integration capabilities while upholding the essence of its trusted features.

1.5 Ataccama Expression Language

To facilitate the integration of Ataccama's data quality rules, a thorough understanding of the Ataccama Expression Language is essential. Below is an overview of the language components, types, and operational logic.

1.5.1 Anatomy of an expression

The structure of an expression in Ataccama's language consists of:

- Statements
  - Variable Assignments
    Variables are assigned values which can include literals, operations, or function calls.

  - Function Definitions

    Optional definitions that encapsulate logic or operations reusable within the expression.

- Resulting Expression

  This is the final part of the expression where the previously defined variables and functions are utilized to compute a result. The outcome of the resulting expression is the output of the entire expression.

1.5.2 Examples of an expression

The two following examples illustrate the structure of an expression in Ataccama's language.

Simple example: Arithmetic Expression

```
a := 10;
b := a * 2;
b + 5
```

In this example, variables a and b are used in statements to set up values that are manipulated in the resulting expression, b + 5, which computes the final output.

Complex example: Digit sum

```
value := replace(trim(input), '-', '');
function digitSum(integer digit) {
    set.sumExp(
        trim(substituteAll("(.)", "$1 ", toString(digit))),
        " ",
        (x) {toInteger(x)}
    )
}
digitSum(value) % 13 == 0
```

This example demonstrates a more complex expression that includes a function definition, digitSum, which is then called in the resulting expression, digitSum(value). The expression is a simplified excerpt from a data quality rule that checks ISIN numbers for validity.

1.5.3 Components of Ataccama Expression Language

Ataccama Expression Language, also called ONE Expressions, organizes data operations and logic into a series of expressions and operands defined by a rigorous structure:

Operands

Operands in Ataccama Expressions are categorized into four main types:

- Literals

  These include numeric, string, or logical constants (e.g., TRUE, FALSE) and the null literal. All keywords are case-insensitive.

- Columns

  Identified by their names, which require square brackets if they include spaces. In multiple input scenarios, columns are specified using dot notation (input_name.column_name). If only one input is used, dot notation can be omitted.

- Set

  Used exclusively with the IN operation, representing a constant expression. Sets can only appear on the right side of the IN operation.

- Complex Expressions

  These may involve various combinations of the above operand types and function calls.

Data Types and Conversions

Operands can be of specific column types such as INTEGER, FLOAT, LONG, STRING, DATETIME, DAY, and BOOLEAN. Ataccama handles type conversions automatically, widening data types as necessary (e.g., INTEGER - LONG - FLOAT) to accommodate operations.

Handling Null Values

The handling of null values aligns with SQL rules, with a notable exception for the STRING data type. In Ataccama, a null string is considered equal to an empty string, impacting how comparisons and operations are performed.

Variables

Expressions in Ataccama can include sequences of assignment expressions followed by a resulting expression, separated by semicolons. The first occurrence of a variable defines its type, with subsequent references needing to conform to this type.

Operations and Functions

Ataccama ONE supports a variety of operations and functions:

- arithmetic functions
- logical functions
- comparison functions

- set functions
- date functions
- string functions
- bitwise functions
- MinMax functions
- aggregating functions
- conditional functions
- conversion functions
- formatting functions
- word set operation functions

The full list of functions and their descriptions can be found in the Ataccama ONE Expressions documentation [8].

1.6 Summary

Let us summarize the goals of the project:

1.6.1 Use of Python

The solution should utilize Python for the implementation of the data quality expression language. This choice is driven by two key reasons. Firstly, Python is widely recognized and utilized among data engineers, which ensures that the tools developed are easily adoptable and integrate seamlessly into existing workflows. Secondly, Python's compatibility with various data pipeline platforms makes it an ideal candidate for ensuring that our solution can be integrated across diverse data environments efficiently, facilitating broader accessibility and practical utility.

1.6.2 Simple API

In the analysis section, we identified data engineers as the primary users of the solution. The project's main objective is to develop an API that is straightforward and intuitive for data engineers. The API should be simple and user-friendly, avoiding overly complex abstractions or advanced design patterns. For instance, object creation should be straightforward, utilizing basic constructors or simple imported functions to minimize complexity and ensure ease of use.

1.6.3 Local execution

The solution should be designed to run locally, allowing data engineers to execute data quality rules directly within their Python environments. This is relevant with respect to the need for ease of use and security. By enabling local execution, data engineers can test and validate data quality rules without the need to access external servers or applications. This approach also simplifies the development process by removing dependencies on external services, ensuring that the solution is self-contained and easily deployable in various environments.

1.6.4 Compatibility with the Ataccama Expression Language

In order to maintain compatibility with the existing Ataccama ecosystem, the Python implementation should be able to execute the same data quality rules as the original Ataccama Expression Language. This includes supporting the same set of functions, operators, and expressions to ensure that data quality rules defined in Ataccama ONE can be seamlessly executed within Python environments. The Python implementation should mirror the behavior of the original Ataccama Expression Language as closely as possible to ensure consistency and compatibility across platforms.

1.6.5 Reasonable performance

To be considered a viable alternative to the original Ataccama implementation and to similar solutions on the market, the Python implementation should be able to handle data quality rules within Python environments efficiently and effectively. The performance of the Python implementation should be within acceptable limits, where a slowdown by a factor of up to 10 times compared to alternatives might be considered tolerable for deployment, but a 1000 times slowdown would indicate serious efficiency issues that could render the solution impractical. By establishing these performance benchmarks, we can validate that the Python implementation meets minimum requirements for real-world applications, ensuring it is a viable alternative for data engineers who require programmatic access to Ataccama's data quality tools. This will be further discussed in the evaluation section.

2 Design

In the design section, we will detail the process of translating Ataccama's expression language into Python, focusing on local execution and user-friendliness. The aim is to allow data engineers to seamlessly integrate Ataccama's data quality rules into their Python workflows. This section outlines the architecture, methodologies, and tools necessary to adapt Ataccama expressions for effective use in Python environments, ensuring the solution is both practical and easy to use.

2.1 Context of data transformations

Data processing pipelines typically adhere to a structured pattern known as ETL (Extract-Transform-Load)[9]2.1, crucial in data management. These pipelines start by extracting data from various sources, which is then transformed through cleaning, enrichment, and aggregation processes before being loaded into a final storage destination.

This thesis specifically concentrates on this transformation step, where data quality rules can be effectively implemented and integrated.

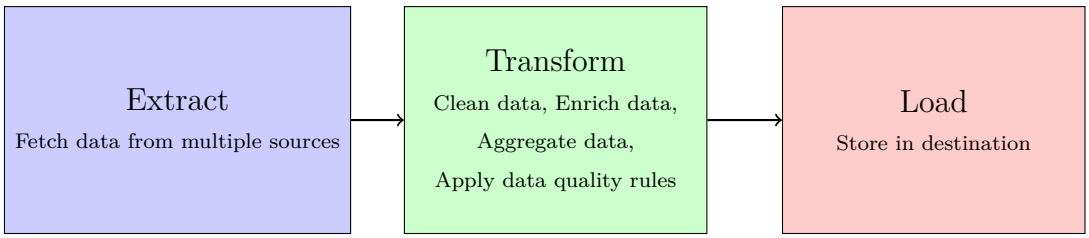

Figure 2.1 Diagram illustrating the ETL (Extract-Transform-Load) process.

2.2 Interface design

The design of the API is straightforward due to the simplicity of the public functionality it offers. Below is a diagram that outlines the public interface of the transpiler:

The interface is uncomplicated, designed primarily to compile and execute expressions efficiently. This simplicity ensures that all essential actions—compiling an expression and executing it—are both intuitive and accessible for users.

The API consists of two primary classes: `Compiler` and `Expression`. The `Compiler` class is responsible for compiling the expression into Python code, while the `Expression` class encapsulates the compiled code and provides a method to execute it with a given record. The record is supplied as a dictionary, where the keys correspond to the field names and the values to the field values.

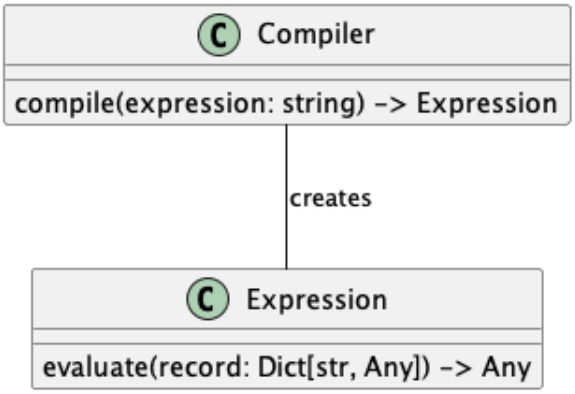

Figure 2.2 API overview

2.3 Architecture

The initial step in developing the solution involves establishing its architecture, which is critical for defining how the program will function.

The architecture of the Python implementation of the Ataccama Expression Language is designed to facilitate the translation of Ataccama's expression language into Python code to enable for local execution, one of the goals of outlined in analysis. This translation process involves parsing the input expression, generating executable code, and evaluating the expression against a dataset. The architecture is structured to accommodate these steps seamlessly, ensuring the efficient execution of data quality rules within a Python environment.

For the first step, the input expression will be converted into an AST (Abstract syntax tree). Given the complexity of the expression language used by Ataccama, employing a parser generator is deemed the most effective approach. The parser generator will utilize the predefined grammar of the language to generate a parser capable of translating input expressions into an AST. This facilitates the incorporation of custom logic in the subsequent steps, particularly during the semantic analysis phase.

During semantic analysis, the AST will be traversed to construct executable code. This transformation is essential for preparing the expression for evaluation in a Python environment, from which the results can be dynamically generated and returned.

The final component of the architecture involves executing the generated code on the provided records. This step ensures that each record is evaluated according to the defined expressions, and the outcomes are systematically returned to the user.

2.3.1 Parser generator

For the parser generator, a specific approach is indicated. The Ataccama Expression Language implementation uses a parser generator called ANTLR. ANTLR (ANother Tool for Language Recognition) is a powerful parser generator[10]. It's widely used to build languages, tools, and frameworks. From a grammar, ANTLR generates a parser that can build and walk parse trees[11]. As the grammar of the Ataccama Expression Language is already defined, it is simple and robust to

use, adapt and reuse the grammar by also using ANTLR to generate the parser.

2.3.2 Code generation

Having decided on the parser generator, the next step is to decide how to generate the code for the expression. There are two obvious approaches at hand: Represent the expression in an object tree with execution being a recursive descent through the tree. The second approach is to generate Python code directly. This can be done using Python standard module `ast`, which can be used build an abstract syntax tree of the expression, and then compile it into a Python function. Alternatively, the code can be generated as a string and then executed using Python's exec function, but this approach is less safe, more error-prone, harder to debug and introduces more overhead as it adds an additional step of parsing the code.

The second approach is more efficient, as it avoids the overhead of traversing the AST, but it is also more complex, as it requires generating Python code. The first approach might appear simpler, but it is less efficient, as it requires traversing the AST and does not include the option to use compilation to Python bytecode.

Using Python as the runtime also comes with the benefit of being able to use Python's scope resolution and name hiding to implement the scoping rules of the Ataccama Expression Language, so a reimplementation can be avoided.

For these reasons, the second approach is chosen. The code will be generated as Python code using the `ast` module, and then compiled into a Python module.

2.4 Compatibility with the Ataccama Expression language

The Ataccama Expression Language is complex and has many features, along with platform specific quirks a peculiarities. For this reason, the implementation will not be a one-to-one copy of the language, but rather a subset of the language features that are most commonly used with some differences in behaviour.

The differences between the Ataccama Expression Language and the Python implementation will be outlined in detail in the following sections. As the goal is to make the implementation as close to the original language as possible, the differences will be kept to a minimum. Consequently, the implementation will be able to run most of the data quality rules written in the Ataccama Expression Language.

To ensure compatibility, the test suite will be created based on the Ataccama Expression Language test suite. The test suite will be used to validate the implementation and ensure that it behaves as expected.

The rest of this section describes a high-level overview of the differences that will have to be introduced along with the reasons behind them. Most of the differences are a result of a different underlying technology, decisions have to be made on where to draw the line between mimicking the original language and

This section has two purposes: The first is to describe the language and its features, the second is to outline and discuss design decisions related to the individual fetatures and functionality that have to be made in order to implement

the language in Python introducing accepting differences for the sake of simplicity and performance.

2.4.1 Dynamic typing

The Ataccama Expression Language is statically typed, following its language of implementation which is Java. This means that the type of each variable and expression is known at compile time compile time. This allows the compiler to catch type errors at compile time, and to generate more efficient code. Also, it allows for function and operator overloading, as the compiler can choose the correct function based on the types of the arguments. This is possible thanks to the record format being known at compile time. The record format is a schema that defines the types of the fields in the record. Python is dynamically typed, which means that it is possible to allow for dynamic typing in the implementation.

On the other hand, to reimplement static typing in Python would require aditional work like keeping track of the types of all symbols and expressions and resolving function overloads.

Furthermore, static typing would require the user to define the record format at compile time, which would make the API less user-friendly, which in our case is a priority. This could be solved by allowing the user to define the record format optionally.

Considering the above stated arguments, the decision is to allow for dynamic typing in the implementation, as it is easier to implement and more flexible. The implementation of optional static typing is a possible future improvement, but for the initial implementation would constitute a great effort for little gain from the user's perspective.

Function and operator overloading

The decision to utilize dynamic typing in the Python implementation of the Ataccama Expression Language carries significant implications for function and operator overloading. Unlike in Java, where Ataccama's static typing enables the compiler to select the correct function or operator based on argument types at compile time, Python's dynamic typing means the types of variables are only known at runtime. This characteristic prevents overloading functions and operators based on type, as there is no way to determine the type of the inputs beforehand.

As a result, each function in the Python implementation must be universally applicable, handling all expected input types through internal logic. This requires implementing comprehensive type checking within each function, where the function determines the appropriate action based on the runtime types of the arguments. Such an approach aligns with Python's duck typing philosophy, where operations are attempted regardless of type, with the function internally managing any type mismatches or errors. This method ensures flexibility and broad applicability of functions, albeit at the cost of the type-specific optimization possible in statically typed languages like Java.

2.4.2 Implementation scope

The Ataccama Expression Language supports over 150 functions but the implementation in Python will be limited to a subset of the language features. The reasoning behind this is that many of the functions are not commonly used, and the implementation of all of them would be a significant effort and would be out of scope for this project as the goal is to provide a simple prototype and validate the viability first.

For this reason, the functions have been categorized by priority, and the implementation will focus on the high- and medium-priority functions. The categorization is based on the frequency of use of the tasks in the data quality rules, and the complexity of the implementation. The high-priority functions are the most commonly used functions, and the medium-priority functions are less commonly used but still important. The prioritization comes from Ataccama's knowledge base. The low-priority functions are the least commonly used functions, and will not be implemented in the initial version of the implementation.

A list of all function along with their priority and implementation status can be found in the appendix A.1.

2.4.3 Arithmetic

The Ataccama Expression Language operates, like standard types in Java, in fixed-size bit arithmetic, i.e., 32 bits for integers and 64 bits for Python. On the other hand, Python uses arbitrary precision arithmetic, which means that the size of the integers is not limited. This means that the results of arithmetic operations can differ between the two languages. For everyday operations, the difference is not significant, and it could be said that the Python behaviour is better. However, in some use cases, the fixed-size arithmetic is necessary, for example, when working with hash tables. As the number of these cases is limited and most of them are provided as implemented functionality, the implementation will use Python-native arbitrary precision arithmetic and handle fixed-size arithmetic as a special case where necessary.

Furthermore, the Ataccama Expression Language provides arithmetic operators for addition, subtraction, multiplication, division, integer division, and remainder. Python provides the same operators, but the behaviour of the operators is different. The first significant difference is the handling of null values. In Ataccama Expression Language, the operators are null safe and follow a SQL-like behaviour. In Python, the operators throw an exception if any of the operands is null. Also, the arithmetics are different; modulo and division produce different results for negative operands. These differences will have to be addressed as the results differ too much to be ignored. Moreover, Python uses arbitrary precision arithmetic, whereas Ataccama Expressions use the underlying Java types, but this difference is not significant for most use cases and could be considered an improvement. Lastly, the plus operator in Ataccama Expression Language is overloaded for string concatenation, converting any non-string to string first, which is not the case in Python. This will have to be implemented as it is a common use case.

2.4.4 Null handling and null coalescing

The way Ataccama Expression language handles nulls has a lot of aspects which have to be addressed.

Operators handle null values in a SQL-like way, mostly returning null if any of the operands is null. The implication for the implementation is that it will not be possible to use native Python operators, as they do not handle null values in the same way.

Empty strings are treated as null values. The documentation states: "A null string and an empty string are considered equal". Moreover, in the Ataccama Expression Language, most empty string returns are coalesced to null. This behaviour also has its own quirks, for example '1 + null == null' but '1 + "" == "1"', which breaks the aforementioned statement.

Furthermore, functions have to be prepared to handle null values in any of the arguments. Most commonly, functions return null if any of the arguments is null, so extensive null checking has to be implemented in the functions. The implementation in Python will have to address these differences and provide a way to handle null values in a Python environment. Date and floating point formatting

2.5 Summary

The design of the Python implementation of the Ataccama Expression Language is structured to facilitate the translation of Ataccama's expression language into Python code for local execution. The architecture is designed to accommodate the parsing of input expressions, the generation of executable code, and the evaluation of expressions against a dataset. The implementation will focus on a subset of the language features, prioritizing high- and medium-priority functions based on their frequency of use and complexity. The implementation will also address key differences between the Ataccama Expression Language and Python, such as dynamic typing, null handling, and arithmetic operations, to ensure compatibility and functionality. The design decisions outlined in this section provide a roadmap for the development of the Python implementation, guiding the translation of the Ataccama Expression Language into a Python environment for accessible and easy data quality rule execution.

3 Implementation

The implementation phase of the project is critical for translating the design into a functional system. This section details the setup of the development environment, focusing on the tools and technologies selected to ensure a robust and efficient development process. The implementation of the individual features is then discussed, providing detailed documentation of the coding process, including snippets and explanations of how the Ataccama Expression language features are implemented in Python. The testing and validation process is also described. Finally, any challenges faced during the implementation are discussed, along with the resolutions that were implemented to overcome them.

3.1 Architecture

The following diagram illustrates the class diagram of the transpiler 3.1.

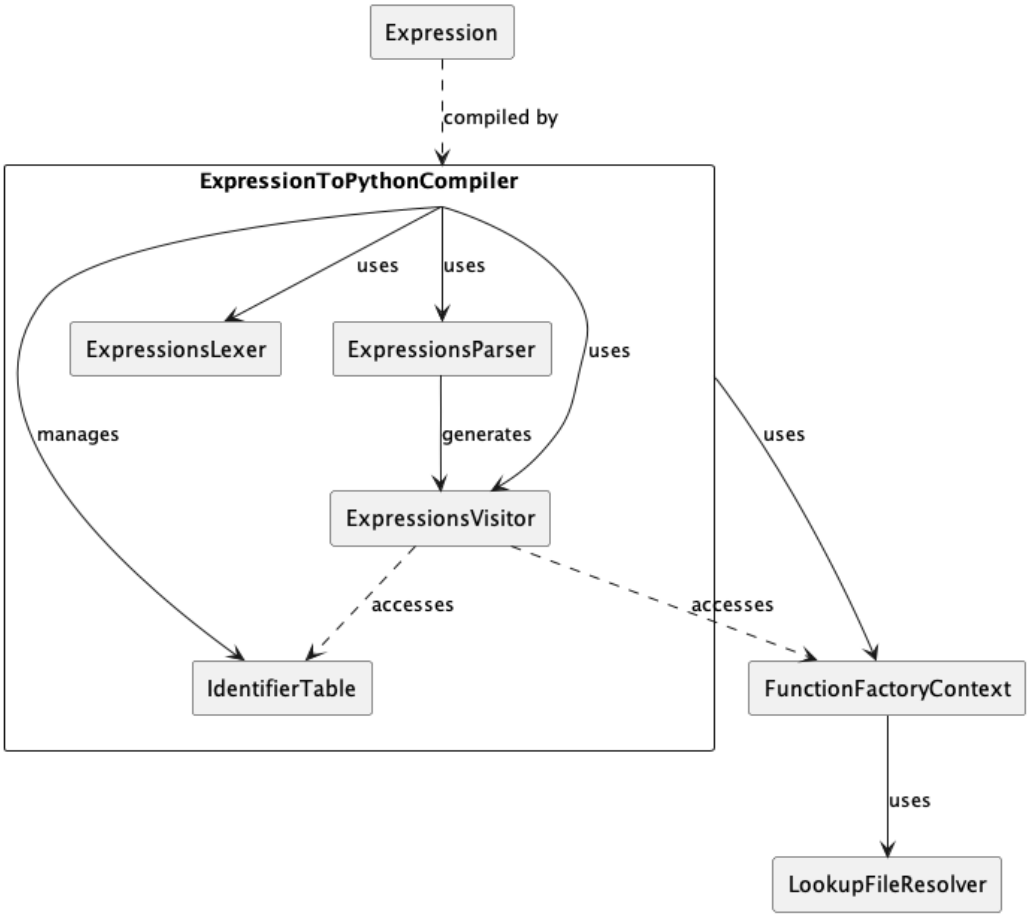

Figure 3.1 Architecture overview

The `ExpressionToPythonCompiler` is initiated with contexts that include external function handling capabilities. It registers various operators and functions that can be used within expressions, ensuring that these are correctly integrated into the compiled code. The lexer and parser transform the input expressions into a format that can be used to generate Python bytecode, with the visitor handling

the actual translation into code and symbol resolution. The option for user to provide custom functions is also available.

The `IdentifierTable` plays a crucial role in managing the names for fields, variables, and functions. The `FunctionFactoryContext` and `LookupFileResolver` provide mechanisms to incorporate external functionalities and data, enhancing the flexibility and power of the expression evaluation.

Finally, the `Expression` class encapsulates the executable code, providing a method to evaluate it with specific inputs. This architecture supports extensibility and customization, key traits for systems requiring dynamic data manipulation capabilities.

3.2 Implementation of individual features

This section delves into the technical specifics of implementing the key features of the Ataccama Expression Language in Python. The primary goal is to accurately interpret and execute the expression rules defined in Ataccama's custom language using Python tools and libraries.

3.3 Expression Parsing

The first step in processing Ataccama's custom expression language in Python is to parse the expressions into a format that can be programmatically analyzed and executed. This is achieved using ANTLR, a powerful tool that generates a lexer and parser based on the grammar used in Ataccama ONE.

ANTLR Lexer and Parser

The lexer reads the raw input text and converts it into a stream of tokens based on the grammar rules defined for Ataccama's language. The parser then takes these tokens and builds a parse tree.

```
sequence:
    command+ expr
    | expr
    ;
command:
    dfunc
    | assign SEMIC
    ;
assign:
    (varName | name) ASSIGN_OP expr
    ;
```

Visitor Pattern

From the parse tree, a visitor is generated—a component that traverses the parse tree. This visitor uses the visitor design pattern to execute operations based on the nodes of the parse tree. For the expression language, the visitor's

primary role is to transform the parse tree into an AST (Abstract syntax tree) using Python's `ast` module, which can then be executed or evaluated in a Python environment. The visitor visits the non-terminals of the grammar and transforms them to `ast` objects or, where needed, some other intermediate objects.

The top-level non-terminal `main` otherwise known as the compilation unit is visited and the visitor returns an `ast` representation of Python module.

The module, when executed, stores the result of the expression in a variable with a predefined name and from which the result can be retrieved.

The following code snippet provides an example of how the visitor pattern is used to handle an AddExpr node in the parse tree:

```
def visitAddExpr(self, ctx: ExpressionsParser.AddExprContext):
    if ctx.left:
        op = ctx.op.children[0].symbol.type
        if op == ExpressionsParser.PLUS:
            name = Operator.ADD.value
        elif op == ExpressionsParser.MINUS:
            name = Operator.SUB.value
        else:
            raise ValueError(f"Unknown operator: \{op\}")
        identifier = self.get_symbol_table()
                .get_identifier_for_name(name,
                                         "internal",
                                         "load")
        args = [self.visit(ctx.left), self.visit(ctx.right)]
        node = self.create_call_node(identifier,
                                     ctx.start.line,
                                     ctx.stop.line,
                                    args)
        return ExpressionArgument(node)

    return self.visitChildren(ctx)
```

Symbol table

There are two layers of names: visible and internal. Internal names are used for internal workings: operators that are functions, multiline lambdas, etc. Visible names are the names that are visible to the user, i.e. callable functions, variables, record fields, etc.

The symbol table is also used to create child symbol tables, which handles the logic of symbol hiding, i.e. when a symbol is defined in a scope, it hides the symbol with the same name in the parent scope.

The use of a symbol table in our system is not primarily due to limitations within Python's name resolution and scoping capabilities. Instead, it addresses specific challenges that arise in our context. Without a symbol table, name resolution would have to occur at runtime, which complicates any future implementation of name existence checks. Additionally, it is good practice to prevent users from

accessing internal names related to operators or multiline lambda functions. By using a symbol table, we can simplify the mapping of complex names, such as converting `math.tan` to a valid identifier such as `math_tan`, while also providing a robust system for detecting and handling name collisions.

Lastly, the symbol table provides an infrastucture for compile-time type checking, should we decide to implement it in the future.

Error Handling

Syntax errors in the expressions are handled using ANTLR's error listeners. These listeners are customized to provide meaningful error messages that help identify and correct syntax issues in the input expressions.

3.3.1 Statements

Handling variables and functions within expressions involves maintaining a symbol table where each variable's name and value are stored. Variables are parsed and evaluated by the visitor, which checks the symbol table to resolve their values during the execution of expressions.

The symbol table keeps track of all symbols used in expressions. It ensures that each variable is correctly declared and used within its scope. Additionally, symbol transformation is employed to prevent collisions and ensure that variable names are unique within the global execution context.

3.3.2 Functions and Operators

Implementing functions and operators in the Python version of the Ataccama Expression Language involves defining Python functions that correspond to each function and operator in the original language.

Each function from Ataccama's language is mapped to a Python function. These functions handle various data types and perform the necessary computations or data manipulations as defined in the Ataccama language specifications.

Each operator is mapped to a function call with the operand(s) as the arguments. This allows for customizing behavior for arithmetic, logical, and comparison operations to closely align with how they function in the original implementation.

The reimplementation of the functions and operators constitutes a significant portion of the work, as the language supports a wide range of operations that need to be accurately translated into Python.

3.3.3 Additional Features and Utilities

In addition to the core features of the Ataccama Expression Language, several utilities and enhancements are implemented.

Command Line Interface (CLI)

A command-line interface is developed to allow users to interact with the expression evaluation engine directly. This CLI provides a simple way to input

expressions and receive the output, making it easier to test and validate the implementation.

Running ‘poetry install‘ will make the ‘evaluate-records‘ script available. ‘evaluate-records‘ can be used to evaluate records on an expression:

```
echo "John,25\nJohn,18" | evaluate-records "name == 'John' and
 age > 20" --record-format "name:STRING,age:INTEGER"
```

Expression Generator

An expression generator is created to produce random expressions based on the grammar of the Ataccama Expression Language. This tool is useful for testing the parser and visitor components.

3.4 Testing and Validation

Testing and validation play crucial roles in software development, particularly when reimplementing data quality rules to verify their performance in Python environments. In this project, we utilize `pytest`, leveraging its parametrization features to rigorously test the implementation of Ataccama's data quality rules in Python.

The use of test parametrization allows for running the same test function with different input values. It is particularly useful in this project for applying a wide array of test scenarios to the implemented functions to ensure comprehensive coverage and that the rules behave as intended across diverse data sets and conditions.

The primary focus during testing is to ensure that the reimplemented rules behave as similarly as possible to the Ataccama Expression Language, this is achieved using unit tests. Tests are designed to validate both typical and edge-case scenarios, ensuring the rules are robust under various data conditions.

The test suite contains more than 1300 tests, covering the wide range of functions, operators, and expressions. These tests are designed to validate the correctness defined by the Ataccama implementation and ensure that the Python version produces the same results in as many cases as possible.

This extensive testing process is crucial for identifying and resolving any discrepancies between the original and re-implemented rules, ensuring that the Python implementation is accurate and reliable for data quality checks. As such it is a key component of the development process, providing confidence in the correctness and efficiency of the reimplementation.

Expected outputs are set with the help of the original Ataccama ONE Expressions Java engine, which serves as a reference for the Python implementation. This ensures that the Python version produces consistent results with the established behavior of the rules in the Ataccama environment. This is aided by the tool for output comparison, which is discussed in the next section.

3.4.1 Tool for Output Comparison

To validate the accuracy of the reimplementation, we have developed a test suite that compares the outputs of our Python implementation against those

generated by the original Ataccama ONE Expressions Java engine. This ensures that our implementation produces results consistent with the established behavior of the rules in the Ataccama environment.

This test suite can be run programmatically using a script which runs the `pytest` test suite; the suite is using a fixture for running any expression that captures it and if enabled, saves it into a file. The script executes the tests and, for a subset of them, runs the same expressions in the Ataccama ONE Desktop environment. It then compares the outputs from both implementations and highlights any discrepancies.

The output can be used to identify any inconsistencies between the original and re-implemented rules, allowing for further refinement and debugging to ensure the rules are correctly implemented.

3.5 Challenges and Resolutions

3.5.1 Date formatting strings

Problem One of the challenges encountered during the implementation was handling date formatting strings in the Ataccama Expression Language. The original language mirrors Java date formatting patterns [12], which are not directly compatible with Python's `datetime` module [13].

Resolution To address the date formatting issue, a mapping between Java and Python date formatting patterns was created. This mapping allows the Python implementation to interpret the date formatting strings correctly and convert them to the appropriate Python format.

To perform this mapping, a separate grammar with lexer and parser was added to parse the original date formatting string into pattern and text tokens. Pattern tokens are then converted using a lookup table.

Related code is located in the `ataccama.expressions.dateformat` package of A.2.

3.5.2 Multiline lambda functions

Problem Another challenge was implementing multiline lambda functions in Python. The Ataccama Expression Language supports multiline lambda functions, which are not directly supported in Python[14]. This required finding a workaround to enable multiline lambda functions in the Python implementation.

Resolution The solution involved defining full-fledged functions instead of using lambda expressions for multiline needs. To integrate these functions seamlessly and avoid namespace conflicts, we employed a symbol table that manages and mangles names dynamically. This approach ensures that all function names are unique and avoids identifier collisions within the Python environment, effectively replicating the flexibility of Ataccama's multiline lambdas within Python's syntactic constraints.

3.5.3 Lookup Files

Problem Ataccama's expressions can perform lookups against reference data stored in proprietary binary formats with sophisticated hashing strategies. This feature is crucial for validating data against predefined sets which are optimized for performance in Ataccama's native environment.

Solution To handle this, the proprietary lookup functionality was reimplemented in Python. This involved developing a method to read and interpret the binary format into a usable form in Python. Additionally, to mimic the fixed-size arithmetic and specialized hashing used by Ataccama, similar algorithms were implemented in Python, ensuring that the lookup performance remains efficient and consistent with the original implementation.

3.5.4 Null Handling

Problem Ataccama's functions and operators are designed to handle null values gracefully, often returning a null or a neutral value when encountering nulls in expressions. This feature is essential for maintaining data integrity and ensuring robust data quality checks.

Solution The Python implementation adopted a similar approach to null handling. Custom operators and functions were developed to replicate the behavior of Ataccama's handling of nulls. For example, custom implementations of addition (+) and other operators were created to return null or appropriate neutral values when encountering null inputs. This ensures that the data quality rules continue to function predictably and effectively even when faced with incomplete or missing data.

This careful replication of functionality ensures that the Python version of Ataccama's data quality rules maintains the robustness and reliability of the original system, adhering closely to its operational logic and data handling practices.

3.6 Development Environment Setup

For this project, a modern and efficient development environment is set up to facilitate the coding, testing, and deployment phases. The environment leverages several key tools and technologies designed to enhance productivity and ensure the quality of the software developed. Below is a breakdown of the core components of the development setup:

3.6.1 Poetry for Dependency Management and Package Publishing

Poetry[15] is utilized as the primary tool for dependency management and package publishing. It offers a streamlined approach to manage libraries and dependencies, ensuring that the project environment is reproducible and consistent across different setups. Poetry simplifies the management of project dependencies,

and its lock file ensures that the same versions are used in every environment, reducing "works on my machine" problems.

Configuration

The `pyproject.toml` file is configured to list all necessary libraries and their specific versions. This file also includes configurations for package metadata, making it easier to package and distribute the final software if needed.

3.6.2 Pypy for Type Checks

Given the dynamic nature of Python, mypy[16] is incorporated to provide optional static type checking. Mypy is an optional static type checker for Python, designed to combine the benefits of dynamic typing and static typing. By annotating Python code with type hints, Mypy can catch many programming errors before they manifest at runtime. It enhances code quality and reliability, especially in large and complex projects where types play a crucial role in the correctness of the program.

Configuration

Mypy is configured to run as part of the continuous integration process, checking type annotations during development. Some leniencies are allowed in the configuration to strike a balance between strict type checking and development flexibility. For instance, certain third-party libraries without type hints and generated code like the lexer and parser might be excluded from these checks to prevent excessive false positives.

3.6.3 Pytest for Testing

Pytest[17] has been selected as the preferred testing framework for this project also due to its support for parametrized testing and test fixtures.

Configuration

Tests are written to cover various cases, from basic unit tests that validate each function's behavior with different inputs to integration tests that ensure that the system components work together as expected. Pytest fixtures[18] are used to set up and teardown test environments, making it easy to manage test states and dependencies.

3.6.4 Additional Tools and Practices

Version Control

Git is used for version control, with a repository hosted on the company GitLab, providing a robust framework for collaboration and version tracking.

Continuous Integration/Continuous Deployment (CI/CD)

CI/CD pipeline is set up using GitLab CI/CD to automate the testing and deployment process. The pipeline is configured to run tests on every commit and deploy the application to a staging environment if the tests pass. This setup ensures that the software is continuously tested and can be deployed automatically to a production environment when ready.

3.7 Example usage

To demonstrate the usage of the Python implementation of Ataccama's data quality rules, we provide a simple example of evaluating an expression on a set of records. The following code snippet shows how to evaluate an expression on a list of records using the Python implementation:

```
records = ...
compiler = create_compiler()
expression_str = ('( NOT ( lower(continent) in '
                  '\{ "asia", "africa", "europe", '
                  '"north america", "south america", '
                  '"oceania", "antarctica" \} ) )')
expression = compiler.compile(expression_str)

for record in records:
    if expression.evaluate(record):
        ...
```

More examples are provided as part of appendix A.2 in the `examples` directory in the form of `jupyter` [19] notebooks with simple demonstrations followed by explanations and commentaries.

3.8 Summary

In the implementation the goal of keeping compatibility was stressed and the Python implementation was designed to closely mirror the behavior of the original Ataccama Expression Language. This was addressed on a problem by problem basis, with each challenge met with a specific solution that ensured the Python version behaved as closely as possible to the original. Furthermore, a wide suite of tests was developed to validate the correctness of the reimplementation.

The goal of enabling local execution so as to avoid the need for a connection to the Ataccama ONE environment was achieved, with the Python implementation providing a standalone solution for evaluating data quality rules.

4 Evaluation

One of the goals outlined in chapter 1 was for the Python implementation of the Ataccama Expression Language to be viable for real-world applications. As a metric for this viability, we have chosen to evaluate the performance of the Python implementation against similar solutions, Soda Core and Great Expectations. This chapter presents the performance testing methodology, the test environment setup, the test cases, the performance analysis, and a discussion of the results.

4.1 Introduction to Performance Testing

4.1.1 Purpose of Testing

Performance testing is crucial in assessing the viability of the Python implementation of the Ataccama Expression Language, particularly in ensuring it can efficiently and effectively handle data quality rules within Python environments, i.e. comparably to similar solutions. This testing is not about matching the performance of similar solutions - Soda Core and Great Expectations - but rather ensuring that the Python prototype is sufficiently efficient for practical use. The aim is to determine if the Python implementation performs within acceptable limits, where a slowdown by a factor of up to 10 times compared to the similar solutions might be considered tolerable for deployment, but a 1000 times slowdown would indicate serious efficiency issues that could render the solution impractical. By establishing these performance benchmarks, we can validate that the Python implementation meets minimum requirements for real-world applications, ensuring it is a viable alternative for data engineers who require programmatic access to Ataccama's data quality tools.

4.1.2 Testing Framework

Tools and Setup

The performance testing utilizes a structured approach where a specific data quality rule—represented in three different formats: an original Ataccama expression, a Soda Core implementation, and a Great Expectations setup—is executed across a range of dataset sizes. This method allows for a direct comparison of how well the Python implementation scales with increasing data volumes, a critical factor in many data engineering tasks.

Methodology

Dataset Sizes Tests are conducted on datasets of varying sizes, starting from 10 records and scaling up to 1 or 10 million records depending on test case complexity. This range is chosen to simulate different real-world scenarios, from small, manageable datasets to large-scale data processing tasks.

Execution Repetition Each test is repeated ten times to ensure consistency and reliability in the results. This repetition helps mitigate any anomalies or

outliers that could affect the accuracy of the performance assessment. The first repetition is considered a warm-up to allow the Python interpreter to optimize the code before the actual performance metrics are recorded. This approach ensures that the performance measurements are based on the optimized execution of the code - while Python is often perceived as an interpreted language that executes code directly from its high-level syntax, in practice, Python first compiles the source code into bytecode, which is a lower-level, platform-independent representation of the code. This bytecode is then executed by the Python interpreter. During the warm-up phase, the Python interpreter can perform several optimizations on this bytecode, such as type specializations, loop unrolling, conditional simplifications, and inline caching.

Process Isolation Each test run is executed in a freshly started Python process to avoid any potential interference from memory leaks, memory layout, residual data, or other artifacts from previous executions. This approach ensures that each test is conducted in a clean state, providing accurate and unbiased performance measurements.

Measurement Metrics The key performance metric collected during the tests is execution time. This metric provides a direct measure of how long it takes for the Python implementation to process the data quality rule on datasets of different sizes, which is essential for assessing the viability and scalability of the Python implementation.

4.2 Test Environment Setup

This section details the hardware and software specifications of the test environment to ensure that the performance results are reproducible and relevant to typical data engineering scenarios.

4.2.1 Hardware Specifications

The performance tests were conducted on a M1 MacBook Pro with the following specifications:

- Processor: Apple M1 10-core CPU
- Memory: 32 GB
- Operating System: macOS Sonoma 14.4.1

4.2.2 Software Configuration

Python Version Python 3.10 is used for all Python-related tests.

Testing Frameworks For execution time measurements, the `timeit` module is used.

4.3 Test Cases

Test Cases Descriptions and Rationale The choice of test cases for evaluating the performance and viability of the reimplemented Ataccama Expression Language in Python is designed to reflect a range of real-world scenarios that data engineers commonly encounter. These test cases are selected to cover a spectrum of complexity, from relatively simple checks to more involved, multi-condition validations that interact with external data sources and complex logic. This selection ensures that the testing not only assesses basic functionality but also gauges the performance under more demanding processing conditions.

There are two test cases selected for the performance evaluation representing different levels of complexity in data quality rule validation. This scope allows for a comprehensive assessment of the Python implementation's performance across a range of scenarios, from basic data validation to more intricate customer data checks. The test cases are designed to simulate common data quality tasks that data engineers might encounter in their daily work, providing a realistic basis for evaluating the Python implementation's efficiency and effectiveness in handling these tasks.

Both of the test cases include enumerating the failed records, which is a common requirement in DQM tasks. This feature is essential for identifying and addressing data quality issues efficiently, making it a key aspect of the performance evaluation.

4.3.1 Test Case 1: Simple Continent Validation

Description This test involves evaluating a relatively straightforward expression that checks if the value of a 'continent' field does not belong to a predefined list of continent names.

Relevance This test case is chosen for its simplicity and its commonality in data validation tasks. It represents typical scenarios where fields within datasets are validated against a fixed set of allowable values. Only a single condition is used wherein lies the simplicity of the case. Testing this case helps verify the Python implementation's ability to handle basic inclusion checks efficiently, a frequent requirement in data cleaning and standardization processes. It also serves as a check for the system's ability to process simple expressions quickly and accurately, possibly revealing any performance overhead and bottlenecks in the compilation process itself.

Expressions Implementation

```
( NOT ( lower(continent) in \{ "asia", "africa", "europe",
"north america", "south america", "oceania", "antarctica" \} ) )
```

Great Expectations Implementation

```
{
  "data_asset_type": "Dataset",
  "expectation_suite_name": "default",
```

```
  "expectations": [
    \{
      "expectation_type": "expect_column_values_to_be_in_set",
      "kwargs": \{
        "column": "continent_lower",
        "mostly": 1,
        "value_set": [
          "asia",
          "africa",
          "europe",
          "north america",
          "south america",
          "oceania",
          "antarctica"
        ]
      \},
      "meta": \{\}
    \}
  ],
  "ge_cloud_id": null,
  "meta": \{
    "great_expectations_version": "0.18.12"
  \}
\}
```

Soda Core Implementation

```
checks for continents:
    - invalid_count(continent_lower) = 0:
          valid values: ["asia", "africa", "europe",
                         "north america", "south america",
                         "oceania", "antarctica"]
```

4.3.2 Test Case 2: Complex Customer Validation

Description This more complex test case. The complexity lies in applying multiple conditions to validate customer data, involving null checks, file lookups, regular expression pattern matching.

Below is the Ataccama Expression Language expression for the test case:

```
(customernumber IS NULL)
OR ( NOT ( isInFile(contactlastname, "surnames.lkp") ) )
OR (NOT isInFile(contactfirstname, "acc_first_names.lkp"))
OR (contactfirstname IS NULL)
OR ( NOT ( isInFile(contactfirstname, "first_names.lkp") ) )
OR (indexOf(upper(email), "NOREPLY") is not null)
OR ( NOT ( matches(@"^[^@]+@[A-z0-9._-]+\.+[A-z._-]+$", email) ) )
OR (email IS NULL)
OR (trim(email) is in {'NULL', 'Null', 'null', '.', ',',
```

```
'-', '_', '', 'N/A', 'n/a', 'na', 'NA'})
```

The expressions references three lookup files, each of which contains a list of valid values for the corresponding field - more than 1M rows in total.

The Great Expectations and Soda Core implementations are omitted for brevity, but they follow the same pattern as the simple continent validation test case.

Relevance This test case is designed to simulate the complex validation processes often required in customer data management, where multiple fields need to be verified against various conditions. It tests the system's capacity to execute multiple, diverse operations — from static reference data lookups to regular expressions and string manipulations — which are typical in scenarios involving data integration and compliance checks. It provides a robust challenge to the system, testing its performance and accuracy under load and complex logic conditions.

4.4 Performance Analysis

For the purpose of this performance analysis, the primary metric that will be evaluated is execution time. Execution time is chosen as the focus metric because it directly impacts the user experience and operational efficiency in real-world applications of the Python implementation of the Ataccama Expression Language. The efficiency with which the system processes data validations directly affects throughput and responsiveness, which are critical factors for data processing workflows.

4.4.1 Expected Outcomes

This analysis aims to validate that the Python implementation, while possibly slower the original implementation and similar solutions, remains within an acceptable range of performance efficiency. If the Python version performs within a factor of up to 10-20 times slower than similar solutions, it may still be considered viable for scenarios where Python's ease of use and integration capabilities and the provided feature set provide significant value over raw execution speed.

By clearly outlining and adhering to this analytical framework, the performance analysis will provide stakeholders with the critical information needed to make informed decisions about the viability and further development considerations of the Python implementation of the Ataccama Expression Language.

4.4.2 Performance Results

Test Case 1: Simple Continent Validation

Table 4.1 shows the execution time comparison for test case n. 1: Continent Validation. The results are also visualized in Figure 4.1.

More important than the absolute execution time is the relative performance compared to the similar solutions. Table 4.1 shows the relative execution times comparison of similar solution relative to our solution for test case n. 1: Continent Validation. The results are also visualized in Figure 4.2.

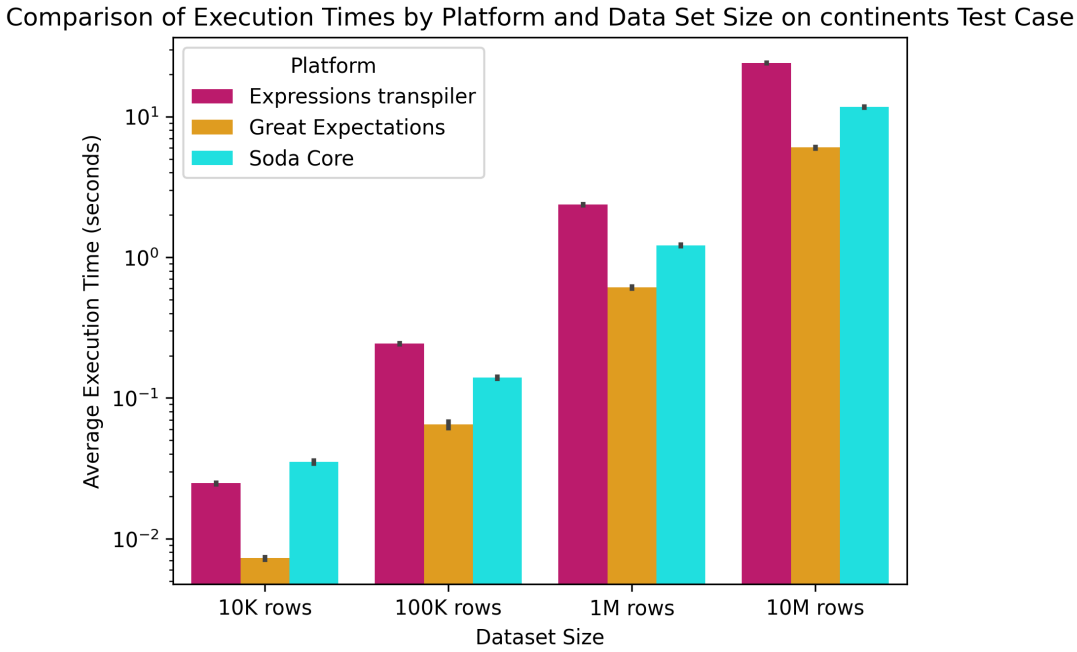

Figure 4.1 Execution Time Comparison for test case n. 1: Continent Validation

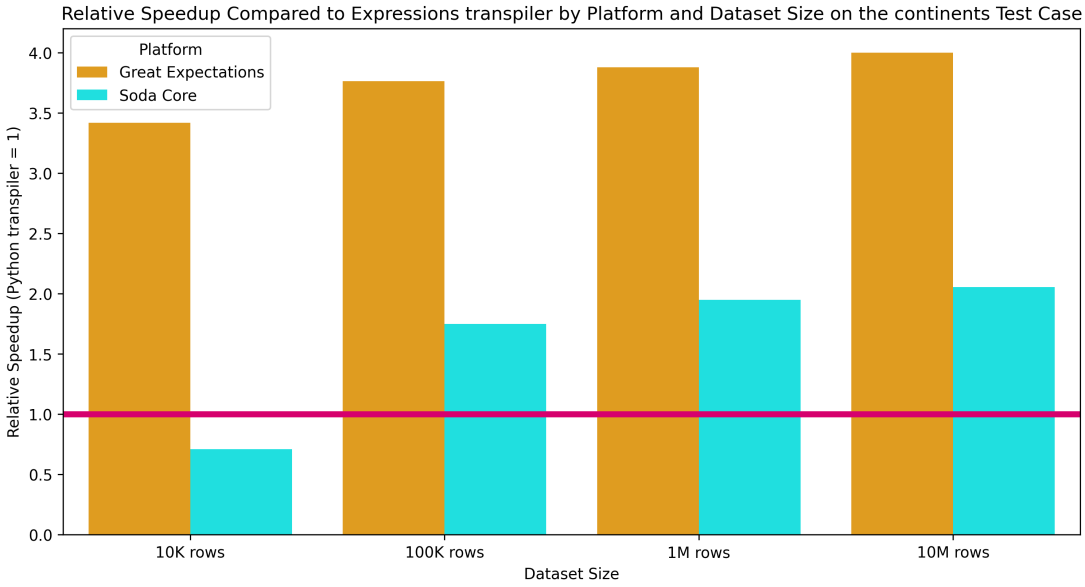

Figure 4.2 Execution times in test case n. 1 of similar solution relative to the Python implementation

data set	platform	mean execution time (s)	standard deviation (s)
10K rows	Expressions transpiler	0.024791	0.000283
10K rows	Great Expectations	0.007251	0.000132
10K rows	Soda Core	0.035015	0.000997
100K rows	Expressions transpiler	0.243883	0.002497
100K rows	Great Expectations	0.064827	0.003331
100K rows	Soda Core	0.139560	0.001877
1M rows	Expressions transpiler	2.373286	0.021965
1M rows	Great Expectations	0.611798	0.009475
1M rows	Soda Core	1.217617	0.015059
10M rows	Expressions transpiler	24.142718	0.167391
10M rows	Great Expectations	6.037510	0.075468
10M rows	Soda Core	11.746598	0.083357

Table 4.1 Execution Time Comparison for test case n. 1: Continent Validation

data set	platform	speedup relative to the Python transpiler
10K rows	Great Expectations	3.418893
10K rows	Soda Core	0.707998
100K rows	Great Expectations	3.762030
100K rows	Soda Core	1.747508
1M rows	Great Expectations	3.879196
1M rows	Soda Core	1.949125
10M rows	Great Expectations	3.998787
10M rows	Soda Core	2.055294

Table 4.2 Relative execution time comparison to Soda Core for test case n. 1: Continent Validation

In this simple test case, even for 10M records, the Python implementation is within an acceptable range of performance efficiency compared to the similar solutions. The execution time is slower than both Soda Core and Greater Expectations, but the difference is no more than 4x.

Test Case 2: Complex Customer Validation

Table 4.3 shows the execution time comparison for test case n. 2: Customers Validation. The results are also visualized in Figure 4.3.

More important than the absolute execution time is the relative performance compared to similar solutions. Table 4.3 shows the relative execution times of similar solutions in comparison to our solution for test case n. 2: Customer Validation. The results are also visualized in Figure 4.4.

In the more complex test case involving reference data lookups, even for 1M records, the Python implementation is within an acceptable range of performance efficiency compared to the similar solutions. The execution time is slower than both Soda Core and Greater Expectations, but the difference is below 10x.

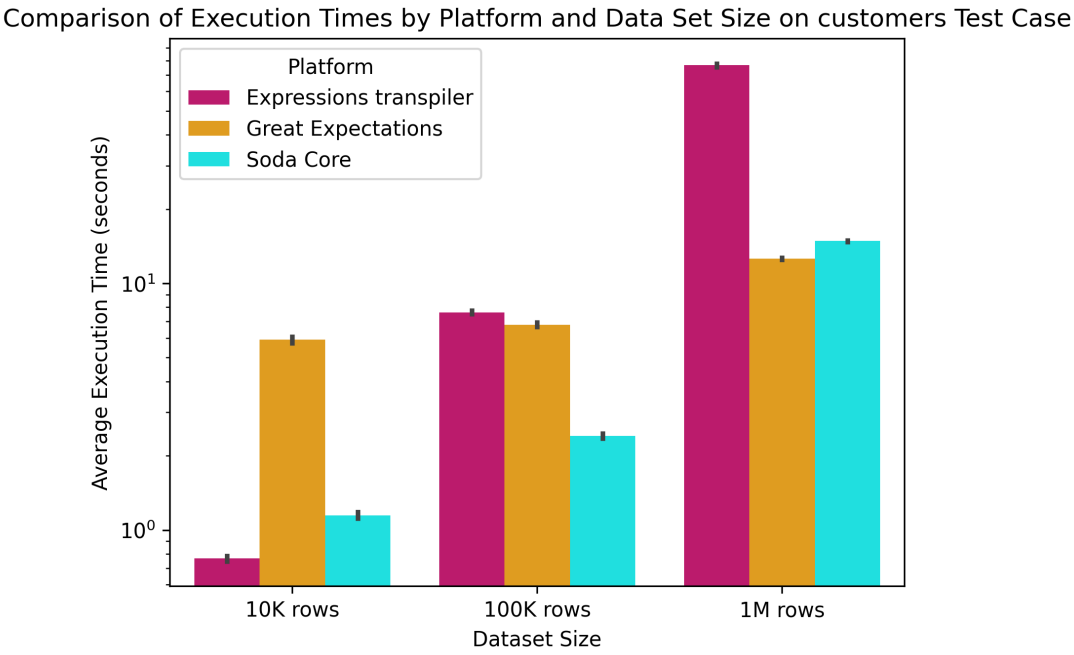

Figure 4.3 Execution time comparison for test case n. 2: Customer Validation

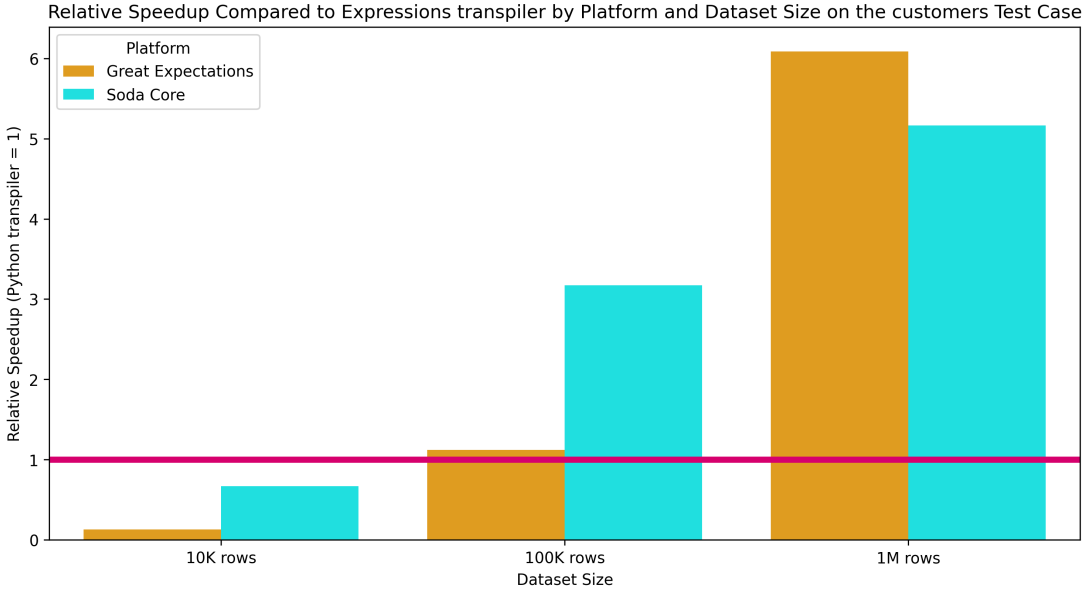

Figure 4.4 Execution times in test case n. 2. of similar solution relative to the Python implementation

data set	platform	mean execution time (s)	standard deviation (s)
10K rows	Expressions transpiler	0.766311	0.018873
10K rows	Great Expectations	5.909839	0.174845
10K rows	Soda Core	1.148130	0.033612
100K rows	Expressions transpiler	7.631781	0.125656
100K rows	Great Expectations	6.799458	0.140569
100K rows	Soda Core	2.405331	0.054677
1M rows	Expressions transpiler	76.628675	1.256378
1M rows	Great Expectations	12.591573	0.095586
1M rows	Soda Core	14.839434	0.100132

Table 4.3 Execution time comparison for test case n. 2: Customer Validation

data set	platform	speedup relative to the Python transpiler
10K rows	Great Expectations	0.129667
10K rows	Soda Core	0.667443
100K rows	Great Expectations	1.122410
100K rows	Soda Core	3.172861
1M rows	Great Expectations	6.085711
1M rows	Soda Core	5.163854

Table 4.4 Execution times in test case n. 2. of similar solution relative to the Python implementation

4.5 Discussion

The performance analysis of the Python implementation of the Ataccama Expression Language reveals that the system is within an acceptable range of performance efficiency for real-world applications. The Python version, while up to 6-times slower on 1M records than similar tested solutions Soda Core and Great Expectations, remains competitive in terms of execution time, even for large datasets. The performance results indicate that the Python implementation can handle data quality rules efficiently and effectively, making it a viable alternative for data engineers who require programmatic access to Ataccama's data quality tools.

The performance gap is acceptable, especially considering the more comprehensive feature set it provides compared to the other two solutions and pluggability with data quality rules in Ataccama.

The Python implementation's performance is likely to improve further with optimizations and enhancements, such as parallel processing, caching, and JIT compilation, which could help narrow the gap with the similar solutions. These optimizations can be explored in the further development of the Python implementation but could also come from further improvements in the Python interpreter itself.

Conclusion

In this thesis, we embarked on the development and subsequent performance analysis of a Python implementation of the Ataccama Expression Language. The objective was to create a programmatic bridge allowing data engineers and analysts to utilize Ataccama's data quality rules within Python.

Analyzing the problem space revealed the need for a flexible and user-friendly solution that could be easily integrated into existing Python-based data processing pipelines while keeping compatibility with the existing Ataccama rule set. This led to the decision to implement the Ataccama Expression Language in Python. For the implementation to be relevant, a performance goal was set to not fall below an acceptable threshold, which was set relative to the execution time of similar solutions.

We designed a solution with a simple API that uses Python's code generation capabilities to translate Ataccama rules into executable Python code. To mantain compatibility with the existing Ataccama rule set, we set an implementation scope and decided to include a wide test suite to ensure the correctness of the translation.

The subsequent phase of the project involved a performance analysis to determine whether the performance goal and as such the practicality of the Python implementation in operational environments was met. This analysis was centered on execution time comparisons with established data quality platforms of Soda Core and Great Expectations, across various dataset sizes ranging from small to large scales. Despite slower execution times in certain scenarios, the results were acceptable. The Python implementation managed to perform within a tolerable slowdown range, typically less than ten times slower than the baseline. This confirmed its viability for scenarios where the ease of integration and the flexibility offered by Python are more critical than the highest possible performance.

This thorough exploration of both development and analysis not only confirms the feasibility of Ataccama's rules in Python but also opens up numerous possibilities for their application in complex data environments.

Bibliography

1. Ataccama. *What is a Data Steward?* [online]. [N.d.]. [visited on 2024-05-09]. Available from: https://www.ataccama.com/blog/what-is-a-data-steward.
2. *Soda Core* [online]. [N.d.]. [visited on 2024-05-07]. Available from: https://docs.soda.io/soda-core/overview-main.html.
3. *Great Expectations Documentation* [online]. [N.d.]. [visited on 2024-05-07]. Available from: https://docs.greatexpectations.io/docs/home/.
4. *Snowflake Documentation* [online]. [N.d.]. [visited on 2024-05-08]. Available from: https://docs.snowflake.com/en/.
5. *Serverless-Datenintegration – AWS Glue – Amazon Web Services* [online]. [N.d.]. [visited on 2024-05-08]. Available from: https://aws.amazon.com/de/glue/.
6. jonburchel. *Azure Data Factory Documentation - Azure Data Factory* [online]. [N.d.]. [visited on 2024-05-08]. Available from: https://learn.microsoft.com/en-us/azure/data-factory/.
7. *Databricks documentation* [online]. [N.d.]. [visited on 2024-05-08]. Available from: https://docs.databricks.com.
8. *ONE Expressions :: Ataccama ONE* [online]. [N.d.]. [visited on 2024-05-07]. Available from: https://docs.ataccama.com/one/latest/common-actions/one-expressions.html.
9. *Was ist ETL? – Extract Transform Load erklärt – AWS* [online]. [N.d.]. [visited on 2024-05-09]. Available from: https://aws.amazon.com/de/what-is/etl/.
10. *ANTLR* [online]. [N.d.]. [visited on 2024-05-08]. Available from: https://www.antlr.org/.
11. *AnTLR 4 Documentation* [online]. [N.d.]. [visited on 2024-05-07]. Available from: https://github.com/antlr/antlr4/blob/master/doc/index.md.
12. *DateTimeFormatter (Java Platform SE 8)* [online]. [N.d.]. [visited on 2024-04-28]. Available from: https://docs.oracle.com/javase/8/docs/api/java/time/format/DateTimeFormatter.html.
13. *datetime — Basic date and time types* [online]. [N.d.]. [visited on 2024-05-07]. Available from: https://docs.python.org/3/library/datetime.html.
14. *6. Expressions* [online]. [N.d.]. [visited on 2024-05-07]. Available from: https://docs.python.org/3/reference/expressions.html.
15. *Introduction | Documentation | Poetry - Python dependency management and packaging made easy* [online]. [N.d.]. [visited on 2024-05-09]. Available from: https://python-poetry.org/docs/.
16. *mypy - Optional Static Typing for Python* [online]. [N.d.]. [visited on 2024-05-09] Available from: https://mypy-lang.org/.

17. *pytest: helps you write better programs — pytest documentation* [online]. [N.d.]. [visited on 2024-05-07]. Available from: `https://docs.pytest.org/en/8.2.x/`.

18. *pytest fixtures: explicit, modular, scalable — pytest documentation* [online]. [N.d.]. [visited on 2024-05-07]. Available from: `https://docs.pytest.org/en/6.2.x/fixture.html`.

19. *Project Jupyter Documentation* [online]. [N.d.]. [visited on 2024-05-07]. Available from: `https://docs.jupyter.org/en/latest/`.

List of Figures

List of Tables

Acronyms

AST Abstract syntax tree 21, 22, 28

DQM Data Quality Management 7, 8, 10, 13, 15, 37

ETL Extract-Transform-Load 13, 14, 20, 47

GDPR General Data Protection Regulation 7

HIPAA Health Insurance Portability and Accountability Act 7

A Attachments

A.1 List of Functions

Table A.1 List of all functions and their priority and implementation status.

Function	Category	Status	Priority
math.sqr	Math functions	Implemented	Low
math.e	Math functions	Implemented	Low
math.log10	Math functions	Implemented	Low
math.tan	Math functions	Implemented	Low
math.sqrt	Math functions	Implemented	Medium
math.exp	Math functions	Implemented	Low
math.sin	Math functions	Implemented	Low
math.atan	Math functions	Implemented	Low
math.pi	Math functions	Implemented	Low
math.acos	Math functions	Implemented	Low
math.asin	Math functions	Implemented	Low
math.log	Math functions	Implemented	Low
math.cos	Math functions	Implemented	Low
math.pow	Math functions	Implemented	Low
removeAccents	String functions	Implemented	High
length	String functions	Implemented	High
jaroWinkler	String functions	Not Implemented	NaN
upper	String functions	Implemented	High
matches	String functions	Implemented	High
lastIndexOf	String functions	Implemented	High
trim	String functions	Implemented	High
capitalize	String functions	Implemented	High
trashNonLetters	String functions	Implemented	High
sortWords	String functions	Implemented	Medium
indexOf	String functions	Implemented	High
diceCoefficient	String functions	Not Implemented	Low
jaccardCoefficient	String functions	Not Implemented	NaN
right	String functions	Implemented	High
soundex	String functions	Not Implemented	Low
transliterate	String functions	Implemented	Low
distinct	String functions	Implemented	Medium
wordCombinations	String functions	Not Implemented	Low
set.sumExp	Set functions	Implemented	Low
replace	String functions	Implemented	High
set.symmetricDifference	Set functions	Not Implemented	Low
containsWord	String functions	Implemented	High
set.lcsIntersectionResultExp	Set functions	Not Implemented	Low
set.lastIndexOfExp	Set functions	Implemented	Low
set.intersectionExp	Set functions	Not Implemented	Low
set.filterExp	Set functions	Implemented	Low
set.lcsSymmetricDifferenceResult	Set functions	Not Implemented	Low
set.distinct	Set functions	Implemented	Low
editDistance	String functions	Implemented	Medium
set.lcsDifferenceResultExp	Set functions	Not Implemented	Low
set.lastIndexOf	Set functions	Implemented	Low
wordCount	String functions	Implemented	High
set.differenceExp	Set functions	Not Implemented	Low
set.approxSymmetricDifference	Set functions	Not Implemented	Low
word	String functions	Implemented	High
set.unionResultExp	Set functions	Not Implemented	Low
left	String functions	Implemented	High
set.indexOf	Set functions	Implemented	Low
set.lcsDifferenceResult	Set functions	Not Implemented	Low
isNumber	String functions	Implemented	High
set.differenceResultExp	Set functions	Not Implemented	Low
set.containsExp	Set functions	Implemented	Low
set.union	Set functions	Not Implemented	Low
levenshtein	String functions	Implemented	Medium
set.lcsSymmetricDifferenceResultExp	Set functions	Not Implemented	Low
set.difference	Set functions	Not Implemented	Low
trimLeft	String functions	Implemented	High
set.lcsIntersectionResult	Set functions	Not Implemented	Low
eraseSpacesInNames	String functions	Implemented	High
set.lcsSymmetricDifference	Set functions	Not Implemented	Low

Continued on next page

Table A.1 List of all functions and their priority and implementation status.

Function	Category	Status	Priority
set.unionExp	Set functions	Not Implemented	Low
trashNonDigits	String functions	Implemented	High
set.indexOfExp	Set functions	Implemented	Low
replicate	String functions	Implemented	Low
set.intersection	Set functions	Not Implemented	Low
set.unionResult	Set functions	Not Implemented	Low
set.distinctExp	Set functions	Implemented	Low
set.size	Set functions	Implemented	Low
set.intersectionResult	Set functions	Not Implemented	Low
set.item	Set functions	Implemented	Low
preserveCase	String functions	Not Implemented	Low
set.symmetricDifferenceResult	Set functions	Not Implemented	Low
hamming	String functions	Implemented	Low
set.subSequence	Set functions	Implemented	Low
set.lcsIntersection	Set functions	Not Implemented	Low
set.differenceResult	Set functions	Not Implemented	Low
set.symmetricDifferenceResultExp	Set functions	Not Implemented	Low
trimRight	String functions	Implemented	High
set.contains	Set functions	Implemented	Low
set.symmetricDifferenceExp	Set functions	Not Implemented	Low
set.intersectionResultExp	Set functions	Not Implemented	Low
set.lcsDifference	Set functions	Not Implemented	Low
ngram	String functions	Not Implemented	Low
trashConsonants	String functions	Not Implemented	Low
set.lcsDifferenceExp	Set functions	Not Implemented	Low
set.lcsSymmetricDifferenceExp	Set functions	Not Implemented	Low
set.lcsIntersectionExp	Set functions	Not Implemented	Low
set.sort	Set functions	Implemented	Low
set.mapExp	Set functions	Implemented	Low
isInFile	String functions	Implemented	High
cpConvert	String functions	Implemented	Medium
substituteAll	String functions	Implemented	High
metaphone	String functions	Not Implemented	Low
doubleMetaphone	String functions	Not Implemented	Low
lower	String functions	Implemented	High
substituteMany	String functions	Implemented	Medium
trashVowels	String functions	Not Implemented	Low
capitalizeWithException	String functions	Implemented	Medium
countNonAsciiLetters	String functions	Implemented	Medium
trashDiacritics	String functions	Not needed	NaN
substr	String functions	Implemented	High
squeezeSpaces	String functions	Implemented	Medium
find	String functions	Implemented	High
is in	Set operations	Implemented	High
in	Set operations	Implemented	High
not in	Set operations	Implemented	High
is not in	Set operations	Implemented	High
bitand	Bitwise functions	Implemented	Low
bitxor	Bitwise functions	Implemented	Low
bitor	Bitwise functions	Implemented	Low
bitneg	Bitwise functions	Implemented	Low
case	Conditional expressions	Implemented	High
decode	Conditional expressions	Implemented	Medium
nvl	Conditional expressions	Implemented	High
iif	Conditional expressions	Implemented	High
getRuntimeVersion	Other operations	Not Implemented	Low
is	Other operations	Implemented	High
is not	Other operations	Implemented	High
setParameterValue	Other operations	Not Implemented	Low
getParameterValue	Other operations	Not Implemented	Low
NOT	Logical operations	Implemented	High
AND	Logical operations	Implemented	High
XOR	Logical operations	Implemented	Medium
OR	Logical operations	Implemented	High
today	Date functions	Implemented	High
dateTrunc	Date functions	Implemented	High
now	Date functions	Implemented	High

Continued on next page

Table A.1 List of all functions and their priority and implementation status.

Function	Category	Status	Priority
dateAdd	Date functions	Implemented	High
datePart	Date functions	Implemented	High
getRequestTime	Date functions	Implemented	NaN
dateDiff	Date functions	Implemented	High
getDate	Date functions	Implemented	High
geoDistance	Uncategorized functions	Implemented	Low
namedSequence	Uncategorized functions	Implemented	Low
coding.fromBase64	Coding functions	Implemented	NaN
randomUUID	Uncategorized functions	Implemented	Low
coding.md5, encode.md5	Coding functions	Implemented	NaN
sequence	Uncategorized functions	Implemented	Low
coding.toBase64, encode.base64	Coding functions	Implemented	NaN
random	Uncategorized functions	Implemented	High
max	MinMax functions	Implemented	High
safeMin	MinMax functions	Implemented	Medium
safeMax	MinMax functions	Implemented	Medium
min	MinMax functions	Implemented	High
xpath	Xml functions	Implemented	Low
toString	Conversion and formatting	Implemented	High
math.round	Conversion and formatting	Implemented	Medium
math.longFloor	Conversion and formatting	Implemented	Medium
math.ceil, math.ceiling	Conversion and formatting	Implemented	Medium
math.abs	Conversion and formatting	Implemented	Medium
toLong	Conversion and formatting	Implemented	High
math.floor	Conversion and formatting	Implemented	Medium
getMilliseconds	Conversion and formatting	Implemented	High
toInteger	Conversion and formatting	Implemented	High
toDate	Conversion and formatting	Implemented	High
toDateTime	Conversion and formatting	Implemented	High
toFloat	Conversion and formatting	Implemented	High
math.longCeil, math.longCeiling	Conversion and formatting	Implemented	Medium
>=	Comparison operators	Implemented	High
=, ==	Comparison operators	Implemented	High
<	Comparison operators	Implemented	High
>	Comparison operators	Implemented	High
<=	Comparison operators	Implemented	High
<>, !=	Comparison operators	Implemented	High

A.2 Expressions transpiler code

The expression transpiler package is the main source code package. It contains the source code, tests, documentation, and additional files.

```
├── src
│   └── ataccama
│       └── expressions ..... main code package
├── tests
│   └── ataccama
│       └── expressions ..... package with pytest test suite
├── examples ..... jupyter notebooks with simple examples
├── docs ..... markdown files with high-level docs
├── pyproject.toml ..... package file
├── README.md ..... overview with installation and usage instructions
├── DEVELOPMENT.md ..... contains information about how to run tests
└── poetry.lock ..... lock file with dependency versions for installation
```

A.3 Performance analysis code

The performance analysis is for reproduction purposes provided as code. The package has following structure:

- `expressions_perf_test` ..... package containing source code
  - `continent` ..... implementation of test case n. 1
  - `customers` ..... implementation of test case n. 2
  - `common` ..... common code for test cases
  - `run_single.py` ..... main script for running a single test case
- `README.md` ..... installation and usage instructions
- `requirements.txt` ..... list of required packages, used for installation
- `run_all.sh` ..... script for running all test cases
- `results.csv`. ..... results of the performance analysis