JOHANNES KEPLER
UNIVERSITÄT LINZ

Author
David Hohensinn, BA

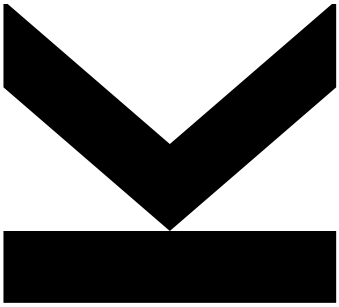
Submission
Department for Business Informatics – Data & Knowledge Engineering

Thesis Supervisor
o. Univ.-Prof. Dipl.-Ing. Dr. techn. Michael Schrefl

Assistant Thesis Supervisor
Mag. Dr. Bernd Neumayr

January 2021

Spooq: A Software Library for ETL Processes in Data Lakes



Master's Thesis

to confer the academic degree of

Master of Science

in the Master's Program

Business Informatics

JOHANNES KEPLER
UNIVERSITÄT LINZ
Altenbergerstraße 69
4040 Linz, Österreich
www.jku.at
DVR 0093696

Statutory Declaration

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources. The submitted document here presented is identical to the electronically submitted text document.

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Masterarbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt bzw. die wörtlich oder sinngemäß entnommenen Stellen als solche kenntlich gemacht habe. Die vorliegende Masterarbeit ist mit dem elektronisch übermittelten Textdokument identisch.

Linz, January 20, 2021 David Hohensinn

Date Signature

Abstract

The implementation of ETL processes in data lakes is a complex and intricate process due to heterogeneous open-source software environments, the use of unstructured data, and the schema-on-read principle. This leads to an increased effort for the development of data pipelines compared to traditional data warehouses, which can rely on years of standards and best practices. The increased development effort affects the duration and quality of data integration projects and can even lead to missed business opportunities. This master thesis deals with the implementation of the software library *Spooq*, which supports data engineers in designing ETL data pipelines in data lakes. The package is based on Apache Spark, which is included in most data lake environments, such as a local Cloudera Hadoop distribution or the cloud-based Azure HDInsight Service. It facilitates testing and documentation and thus enhances the quality of data pipelines. The software library allows data engineers to focus on business logic rather than software code by abstracting Spark's low-level functions. The use of *Spooq* results in reduced development effort for data pipelines.

Kurzbeschreibung

Die Implementierung von ETL-Prozessen in Data Lakes ist aufgrund heterogener Open-Source-Softwareumgebungen, der Verwendung unstrukturierter Daten und des Schema-on-Read-Prinzips ein komplexer und komplizierter Vorgang. Dies führt zu einem erhöhten Aufwand für die Entwicklung von Datenpipelines im Vergleich zu traditionellen Data Warehouses, die sich auf jahrelange Standards und Best Practices stützen können. Der erhöhte Entwicklungsaufwand wirkt sich auf die Dauer und Qualität von Datenintegrationsprojekten aus und kann sogar zu verpassten Geschäftsmöglichkeiten führen. Diese Masterarbeit befasst sich mit der Implementierung der Softwarebibliothek *Spooq*, die Dateningenieure beim Entwurf von ETL-Datenpipelines in Data Lakes unterstützt. Das Paket basiert auf Apache Spark, das in den meisten Data Lake Umgebungen enthalten ist, wie zum Beispiel einer lokalen Cloudera Hadoop-Distribution oder dem cloudbasierten Azure HDInsight Service. Es erleichtert das Testen und Dokumentieren und steigert so die Qualität der Datenpipelines. Die Softwarebibliothek ermöglicht es Dateningenieuren, sich auf die Geschäftslogik statt auf Software-Code zu konzentrieren, indem sie die Low-Level-Funktionen von Spark abstrahiert. Die Verwendung von *Spooq* führt zu einem reduzierten Entwicklungsaufwand für Datenpipelines.

Conventions Used in This Thesis

The following typographical conventions are used in this thesis outside of figures, tables, and code blocks:

Italic

Italic typesetting is used to emphasize important terms and accentuate the artifact's name of this thesis, *Spooq*. Furthermore, this type setting is applied to software library names, values in the context of an attribute or variable, directory names, and URIs.

"Italic in quotes"

Text that describes a rule in the sense of inference is displayed as italic, surrounded by quotes.

`Constant width`

Inline source code and program listings are set in a monospace font. References to code elements, like object or attribute names, are formatted the same.

`Constant width bold`

Commands that are to be executed literally are set in a bold monospace font.

Contents

List of Figures

List of Tables

List of Code Blocks

Part I.

Introduction and Methodology

1. Introduction

More and more data is being generated each day. The era of big data has started years ago and is still expanding rapidly. An IDC white-paper by Reinsel et al. (2018) predicts that the annually generated data will increase from 33 Zettabytes (2018) by 430 percent to 175 Zettabytes by 2025. A run of CERN's Large Hadron Collider produces 25 GB/s, which is more than the information of 35 compact discs per second. (CERN, 2019)
The availability of such a vast volume of data changed how organizations see and utilize this information. New applications and use cases for big data have emerged.

The trend goes towards generating additional value from data in comparison to static analyses. Reporting and Business Intelligence use cases are still valid and heavily applied, but companies start to venture into new territories. May it be customized advertisement for customers, smart search engines based on the previous behavior of similar users, or machine learning applications that can personalize the user's experience for a given product – digital or analog.

Baesens (2014) argues that data analytics and data-driven business designs are more prominent than ever and will continue to grow as data has strategic value, which can be used as a competitive advantage. He specifies various business fields like marketing, risk management, government, web, or logistics as beneficiaries for contained value in data. The exemplary use cases he describes are going from retention modeling, customer segmentation, credit risk modeling over fraud detection, social security fraud, terrorism detection to supply chain analytics, business process analytics, and demand forecasting.

Anyone who has ever worked with real-world data knows that these tasks require a lot of effort and expertise. To be able to generate information from data, it needs to be in an analyzable form. Data comes in all forms and sizes with varying quality. To ease the process of preparing data, a lot of technologies and best practices have evolved.

Data warehouses, NoSQL databases, and data lakes based on distributed file systems, are some of the best known architectural designs which help to manage big datasets. Dimensional data warehouses form the most mature and established architecture, which accounted already in 1995 for two billion dollars revenue, according to Chaudhuri and Dayal (1997).

Data warehouses are often based on RDBMS (Relational Database Management Systems), which can have strict limitations on the volume of data they can store and process depending on the database engine used. Especially the implementation of centralized data warehouses prohibits data volume to scale above a certain maximum depending on the server's hardware. There are, however, alternatives that are not restricted by data volume like Google's *F1*. (Hajmoosaei et al., 2011; Pasupuleti & Purra, 2015; Shute et al., 2012)

Another characteristic of RDBMS-based data warehouses is the strict enforcement of schemata in the ingestion step. This slows down the extraction implementation process and limits the types of data which can be stored and used. (Fang, 2015; Pasupuleti & Purra, 2015)

Szalay and Blakeley (2009) state in their book "The Fourth Paradigm: Data-Intensive Scientific Discovery" that for data-intensive computing in scientific research, RDMBS — like Microsoft's SQL Server — work very well in the range from a few to tens of Terabytes. For bigger amounts of data, the usage of massively parallel computation engines like MapReduce is necessary.

NoSQL (sometimes referred to as "Not Only SQL") databases are easier to horizontally scale, which removes the limitations forced upon by a single server's maximum disk space and memory inherent for centralized database systems. This entails disadvantages for the

transaction support, which brings up problems for the consistency and multi-tenancy across the distributed network. (Jing Han et al., 2011; Sakr et al., 2011)

The schema-less approach of NoSQL databases makes them attractive for ingesting and persisting unstructured data, which relates to one of the four V's (Variety) of big data. (Sadalage & Fowler, 2012) Data lakes are the trending solutions to cope with the amount of big data. A data lake is — according to Fang (2015):

> "... a methodology enabled by a massive data repository based on low-cost technologies that improves the capture, refinement, archival, and exploration of raw data within an enterprise."

This concept is more similar to a distributed file repository than to a database or a document store. The main advantage of data lakes is the possibility to have a single point of truth with no limitations on data types or structures for comparably low costs. The open-source project Apache Hadoop is generally seen as the most mature and widely utilized platform to implement a data lake. (Fang, 2015; Ravat & Zhao, 2019; Sharma, 2018)

Processing data within data lakes is, however, a complex and complicated task that can not be easily standardized due to its openness, both in data structure and computation software. The restrictions posed by the application of RDBMS, through the necessity of schema definitions at the earliest stage of the ETL (Extract, Transform, and Load) process, eases further processing significantly. The maturity of RDBMS-based data warehouses helped to establish frameworks and standards which often reduce the complexity of ETL processes. The advantage of the data lakes' inherent principle of sourcing data in its rawest form speeds up the extraction process substantially but entails more effort in later phases due to the late binding or schema-on-read approach. (Fang, 2015; Pasupuleti & Purra, 2015)

Combining various data formats, structures, and content with several use cases, which need diverse processing concepts themselves, results in a multitude of different transformation process chains. Support for

ad hoc queries has different requirements on software than streaming or batch-processing. Each use case-specific software application also has to be able to handle the concerned data types and formats. (Sharma, 2018)

Every company has its preferences, rules, and necessities for a particular software stack like vendor contracts, available skill sets of its engineers, or restrictions due to the infrastructure. Even though there are often a lot of redundant logic steps and reusable processes in the processing pipelines, there is no one-size-fits-all solution.

Extracting data from various sources, transforming it appropriately for any given use case, and loading it to an accessible location with high query performance poses a complex process. Kimball and Caserta (2004) argue that this process — also called Extract, Transform, and Load (ETL) — easily accounts for 70 percent of the effort needed to implement and maintain a typical data warehouse. Data lakes serve a superset of goals, including the main goals of data warehouses like reporting and enabling business users to work more data-driven. The major difference between the two is when the ETL processes are performed in the data life cycle. (Fang, 2015; Pasupuleti & Purra, 2015)

Data scientists and business analysts are often not able to do the necessary data transformation steps on their own and need the help of a data engineer. Developing, testing, and deploying ETL pipelines is done by engineers, which takes much time due to the complexity. The longer the data ingestion and preparation takes, the longer it takes to gain useful information and consequently generate value out of it. (Anderson, 2019)

How to treat and transform data is highly dependent on the context of the use case and the data itself. Ravat and Zhao (2019) emphasize that keeping the metadata management on a high quality is especially crucial for data infrastructures, based on data lakes. Inter- and intra-metadata (information about the connections between datasets and knowledge about a specific dataset) provides interested users and

systems the information they need to find, understand, and utilize appropriate datasets. (Ravat & Zhao, 2019)

Semi-Automatic ETL configuration, supported by metadata and a reasoning engine, can ease the process but needs to be adapted carefully to the processing framework. Metadata support is best practice for traditional data warehouses but can not be easily applied to data lakes due to the lack of standardized ecosystems. (Kimball & Caserta, 2004)

The current data lake software stack is exceptionally open and powerful but lacks integration within itself. Many best practices from previous architecture designs like data warehouses do not apply due to the fundamental change in data and software heterogeneity. Customized ETL processes allow any kind and every size of data to be ingested, analyzed, and utilized. Standardized procedures and best practices are still missing, which would enable developers to implement data engineering pipelines in a short time with low effort and high quality.

The outcome of this thesis is a software library that wraps around proven technologies to provide reusable code modules. Those will be parameterized with the help of information about the use case context and other metadata. The usage of this software library improves data pipeline development by utilizing ready-made code such that quality improves and implementation effort decreases in order to be able to generate more value in a shorter amount of time.

This thesis describes the planning and execution of a software research and development project with *Spooq* as the main result. The project is based on an evolutionary prototyping approach, augmented by concepts and methods from the Design Science Research Methodology by Peffers et al. (2007).

2. Methodology

This section introduces the reader to the academic context of the problem, the course of action for this software research and development project, and its proposed solution. An iterative software development method, called evolutionary prototyping, combined with principles and activities of the Design Science Research Methodology by Peffers et al. (2007), was applied and will be discussed in more detail.

Business intelligence and data engineering are parts of information systems that form a meaningful and relevant research field in academia and the private sector. ISR (Information Systems Research) consists of multiple approaches that follow different paradigms. There is a broad spectrum of methods an IS (Information Systems) researcher can choose from. The author chose an iterative software engineering method called *evolutionary prototyping*. The development process was enriched by activities and concepts from the Design Science Research Methodology proposal by Peffers et al. (2007). The following sections explain to the reader why a design-oriented research approach was chosen, what this entails, and which steps and actions were carried out.

Gauch Jr (2002) state that scientific research has to adhere to general principles to "increase productivity and enhance perspective" and not to follow a "fixed sequence of steps". Those principles define which methods are appropriate for scientific work instead of presenting a plan on how to do research. A method is commonly understood as a deliberately applied process that brings someone closer to a goal. Scientific methods have additional criteria. They have to be generic (applicable by different actors), logical (reproducible and

validate-able), and effective (reaching a certain goal). (Wilde & Hess, 2007)

The knowledge objects of ISR are information systems in economy and society, both for organizations and individuals. Those systems consist of personal, mechanical, and organizational service providers, as well as of their interactions and inter-dependencies. The knowledge of IS exists partly in scientific literature and primarily in software, organizational solutions, and tool-sets. The main goal of ISR drives strategies and tactics for information systems and innovations (instantiations). (Hevner et al., 2004; Österle et al., 2010)

Most well-known methods, applied in ISR, can generally be split into two epistemic paradigms: *design-oriented* and *behavior-oriented* research. The objective of IS research is to enable the development and implementation of technology-based systems by generating knowledge to solve important business problems. Behavioral science accomplishes this goal by constructing and validating theories which describe or predict phenomena. Design-oriented science, in contrast, changes the phenomenon itself by innovations in the form of artifacts. This makes the main object of knowledge in design-oriented research the construction of artifacts, while behavioral theory takes the world as-is. (Hevner et al., 2004; Österle et al., 2010; Wilde & Hess, 2007; Winter, 2008)

The goal of this thesis is to create an artifact in the form of a software library that can be directly applied to mitigate certain common problems with ETL processes in data lakes. The artifact is incrementally built with broad applicability in mind, in terms of audiences and use cases. The design-oriented software engineering method *evolutionary prototyping* was used for the implementation, which shares the iterative approach of agile software development outlined in the Agile Manifesto by Beck et al. (2013). Iterative processes are also inherent to design science, as stated by Hevner et al. (2004) in their guidelines. However, the differentiation between a professionally implemented software project and a proper design artifact can be difficult and will be addressed in Section 2.3.

Nevertheless, evolutionary prototyping is compatible with general software engineering projects and design science research and is examined in more detail in the next section.

2.1. Prototyping-Oriented Software Development

Iterative software development projects differ from those which use the well-known waterfall model. This section outlines iterative software development frameworks based on prototyping. Evolutionary prototyping, which was used to develop the IT artifact of this thesis, is described in more detail.

One of the most applied best practices for designing software products is the waterfall model. It splits the complex design of a problem-solving software into several smaller phases. Those phases are outlined, implemented, and evaluated in a clear sequential process. The planning of the implementation phases are derived from an explicit and extensive collection of specifications by the customer. The acceptance test by the end-user happens in the last phase of a software's life cycle. Precursory requirements by the customer consist of criteria constructed from the perspective of a closed system where no information about the implementation details is known or considered. Pomberger et al. (1991) state that defining sufficiently exhaustive system designs in advance, which comply with all preliminary requirements, rarely happens in reality due to its complexity and bidirectional dependencies between implementation phases. Learnings and uncovered design issues from later phases in the software life cycle often require changes to previously implemented parts of the software. (Pomberger et al., 1991)

Iterative software development shares most aspects with waterfall model-based engineering. The main difference is that strategies are based on incremental development instead of strict sequential pro-

cesses. Implementation phases are referred to as activities as they can overlap in the perspective of time. Generating demonstrable products on a regular basis allows to evaluate the requirements by the user earlier on in the process, instead of relying solely on the predefined specifications. Through feedback and reflection by the user, the system can be specified, designed, and developed in parallel. Iterative engineering decreases the risk of bad design decisions, incorporates acquired knowledge in the process, and increases the probability of meeting the needs of the customer in functionality and quality. (Pomberger et al., 1991)

Prototyping-oriented engineering paradigms provide frameworks that incorporate iterative processes and therefore take advantage of the benefits mentioned above. Bischofberger and Pomberger (1992) classify prototyping approaches into three different areas based on existing literature:

Exploratory Prototyping
: The focus of exploratory prototyping is mainly on iteratively refining the requirements of the customer to match the expectations as well as possible. Prototypes are to be generated quickly and cheaply to show or simulate the interface of the application and how it would fulfill certain specifications. The frequent integration of the end-user early on produces precise and exact requirements definitions. (Bischofberger & Pomberger, 1992)

Experimental Prototyping
: Improvements to internal interactions and dependencies between system elements form the goal of experimental prototyping. Rather than evaluating the user-facing functionality, experimental prototyping-based software development concentrates on holistic system architecture and its underlying components. Focusing on the interrelations of the individual system's parts allows for a better definition of the architecture. (Bischofberger & Pomberger, 1992)

Evolutionary Prototyping
: Prototypes created with the evolutionary prototyping approach can be viewed as products ex-ante. The development process

targets an incremental software implementation which reuses previous iterations. The constant output is reviewed by the end-user and allows a constant refinement and extension of the required functionality. The incipient system architecture should be designed to allow relatively easy refactoring and amelioration to enable an evolution of the software without major redesigning. The last iteration of the development process, validated by the user, represents the final product of the engineering work. Avoiding throwaway prototypes is the main distinctive feature of evolutionary prototyping compared to other prototyping-oriented development methods. (Bischofberger & Pomberger, 1992)

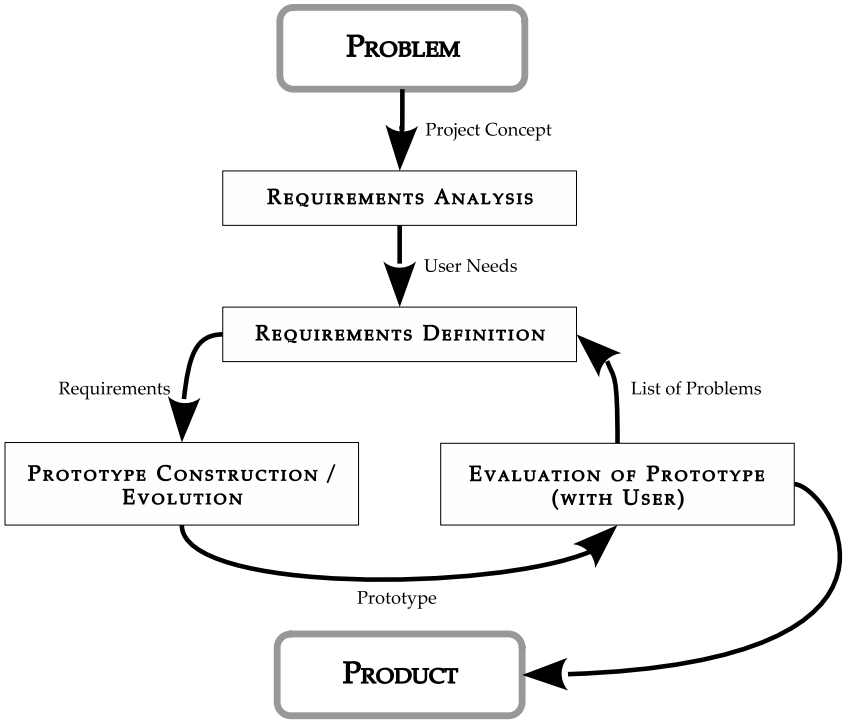


Figure 2.1.: The Evolutionary Prototyping Software Life Cycle - Based on Figures Provided by Bischofberger and Pomberger (1992)

Figure 2.1 shows a typical process model for projects based on evolutionary prototyping. A problem-initiated analysis is followed by the definition of requirements, which presents the entry point to a

circular workflow. A working implementation of previously defined specifications is evaluated with the help of the end-user, which produces a list of problems and discrepancies. This output is then used to extend or overhaul the requirements definitions, leading to another increment of the prototype. An evolutionary prototype is considered a final product when the evaluation certifies compliance with the specification and requirements of the demanded functionality.

Evolutionary prototyping, as a software development method, provides guidance for the implementation of an IT artifact within the scope of a thesis. However, other essential components for academic projects are not targeted by it. Research methodologies, inspired by design science, explicitly focus on activities around the engineering process like problem identification or communication. The next section focuses on design science and describes suitable, non-engineering activities relevant to this thesis.

2.2. Design Science

> "Design science creates and evaluates IT artifacts intended to solve identified organizational problems." (Hevner et al., 2004)

The first reference to use design-oriented methods to solve scientific problems is probably by F. Zwicky within "Morphological Method" in 1948, as stated by Cross (1993). The *Conference on Design Methods* in 1962 is generally seen as the start of using design methodology for substantial academic research. Design methods have their origins in the 50s and 60s after the Second World War to cope with the pressuring problems with the help of novel scientific methods. (Cross, 1993)

Design-focused research methodologies in IS-related fields are mainly applied by researchers of German-speaking countries. Although, design science as a form of design-oriented research is also becoming more accepted and utilized in the Anglo-Saxon area where the be-

havioral research paradigm is still the dominating school of thought. *Wirtschaftsinformatik* is the most similar and closest field of research to IS in German-based academia. It literally translates to *Business Informatics* but is mainly translated to *Information Systems* or *Business & Information Systems Engineering*. (Chen, 2011; Österle et al., 2010)

One of the most cited papers about design science called "Design Science in Information Systems Research" was written by Hevner et al. in 2004. It formulates guidelines for solving problems with design science. Numerous papers rely on those guidelines, which makes them a well-established foundation of design science. The following paragraphs describe the seven guidelines in more detail.

Guideline 1: Design as an Artifact

The outcome of design science research must be an artifact. This can be an instantiation of a software but also constructs, models, and methods, which are applicable to create, operate, or maintain information systems. Artifacts are innovations that define thoughts, processes, abilities, and products to work effectively and efficiently with information systems. (Hevner et al., 2004)

Guideline 2: Problem Relevance

Solving relevant problems for communities of interest through the construction of an artifact is the primary purpose of design science research. The relevance of the problem is, therefore, directly connected to the benefits of solving the problem at hand. This can be better medical diagnostics for the people in general, less wasted resources for ecologically oriented institutions, or simply higher efficiency for work processes and, therefore, higher return on investment for companies. If a *problem* does not have any negative implications for any environment, and consequently no positive effects when it is solved, then it does not fall into the definition of a problem according to design science. (Hevner et al., 2004)

Guideline 3: Design Evaluation

Design artifacts are created to achieve objectives, which are de-

rived from relevant problems. The evaluation of such artifacts indicate if those objectives are met when the solution is applied within the concerned environments. Multiple aspects can be evaluated. *Requirements Fulfillment:* Indispensable prerequisites and constraints have to be complied with, to benefit the environment in which the problem lies. *Effectiveness:* The effect of the application of the artifact relates to achieving the goal itself. *Utilization:* The community of interest, from which the problems stems, must be able to utilize the artifact accordingly. *Quality:* Reliability and maintainability are essential factors for any implementation, as well as efficiency. *Others:* There are further dimensions to be considered for evaluation. For example, the style of the solution, costs of implementation, or acceptance of its clients. Depending on the aspects to evaluate, the environment, available data, and appropriate metrics have to be determined beforehand. Several well-known methods can be used to evaluate the artifact upon its criteria fulfillment rigorously. (Hevner et al., 2004)

Guideline 4: Research Contributions

Design science research must yield novel and engaging contributions, which can be applied in relevant areas, where they show themselves beneficial. These contributions can be categorized into three different types. *The Design Artifact:* The artifact itself solves prior unsolved problems and produces value, when employed in concerned environments. This can be done by enriching the current knowledge base or by utilizing available knowledge in unprecedented fashions. *Foundations:* Other significant contributions to design science research are extensions or enhancements of present foundations like constructs, models, methods, or instantiations. *Methodologies:* Metrics and measures for evaluation scenarios in design science research are essential elements. They help to explain and predict, for example, processes, implementations, or usability. *Implementability* and *representational fidelity* ensure that the artifact represents the environment, in which they are expected to be used, and that it

fulfills the requirements and constraints of those environments, to be practically applicable. (Hevner et al., 2004)

Guideline 5: Research Rigor

The creation and evaluation of design science artifacts must be conducted rigorously, with scientific methods, appropriate for the situation. A. Lee (1999) notes that too much rigor can harm the relevancy of a research's outcome. However, Applegate (1999) and Hevner et al. (2004) are in agreement that adequate rigor and relevancy for research in IS are crucial and not mutually exclusive. Rigor must be applied with respect to the practicability and reproducibility for the creation of design artifacts. Researchers must frequently re-evaluate the relevancy of their evaluation criteria and methods. (Hevner et al., 2004)

Guideline 6: Design as a Search Process

The research process in design science is a continuous search process by definition. The goal is to find an effective solution to an identified problem. Hevner et al. (2004) phrase the thoughts about problem-solving by Simon (1996) that it "... can be viewed as utilizing available means to reach desired ends while satisfying laws existing in the environment." *Means* stand for the entirety of options, which potentially lead to a solution. The objectives of a research project constitute the *Ends*. *Laws*, like in non-research related fields, are inevitable circumstances that can not be changed. Evaluating all possible means to construct a solution meeting the ends and laws, is often not viable or even outside the bounds of possibility due to the capital magnitude of the solution space. Design science interprets satisfactory or *satisficing* (Simon, 1996) solutions as successful research results, which connotes that not necessarily all imaginable alternatives have to be considered, if the solution is effective. (Hevner et al., 2004; Simon, 1996)

Guideline 7: Communication of Research

The developed artifact and the process of research have to be communicated to interested parties in affected fields and com-

> munities. Audiences of interest can be technical- or managerial-oriented. Chief executives of organizations and officials of communities are most interested in the importance and relevancy of the problem itself. Management needs to be able to assess if the solutions can be applied within their organizational structure. They do not need to understand the artifact and its principle of operation in detail. Technical-oriented audiences need more detail about the artifact's functionality to be able to utilize it in their environment. Information about the design science research process allows other researches to build their work on its basis. It also allows for independent reproduction and evaluation of the results. Management needs to know if and how well an artifact can solve a specific problem in their organizational situation while technical persons need to know how to apply and take advantage of it practically. (Hevner et al., 2004)

The seven guidelines presented by Hevner et al. (2004) represent criteria that must be met when conducting research with respect to design science. Most design science-based IS research complies with those rules, but the lack of a generally established framework and methodology impedes fast adoption. Peffers et al. (2007) designed a design science research methodology (DSRM) to facilitate conducting, demonstrating, and assessing design science research.

A methodology is defined as "...a system of principles, practices, and procedures applied to a specific branch of knowledge." (DM Review and SourceMedia, 2019). The framework by Peffers et al. (2007) is consistent with prior knowledge — e.g., the seven guidelines posed by Hevner et al. (2004) — and provides a mental model as well as nominal processes. Their design science research methodology consists of six activities that are visualized in Figure 2.2 by showing the process model within a nominal sequence.

The following list describes each activity of the design science research methodology proposed by Peffers et al. (2007) in more detail.

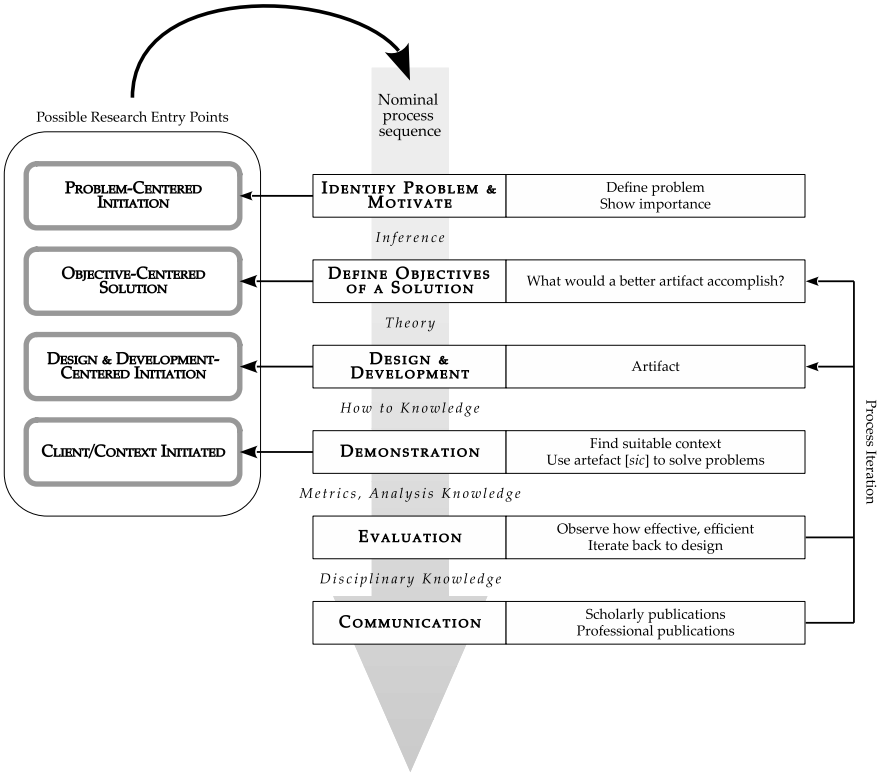


Figure 2.2.: Design Science Research Methodology (DSRM) Process Model - Based on Figures Provided by Peffers et al. (2007)

Activity 1: Identify Problem and Motivate

Identifying and defining the problem, which the artifact addresses, is the essence of the first activity. This corresponds to the second guideline shown above. Confining the problem and its context may demonstrate to be useful, as it allows the artifact to address its complexity more accessibly. Supporting the solution's value motivates both the researcher and the stakeholders. The researcher needs knowledge of the problem conditions for this step, as well as a clear understanding of the importance of the problem's solution. (Peffers et al., 2007)

Activity 2: Define Objectives of a Solution

The problem definition often provides a basis for the objectives,

as goals can be directly inferred from it. The goals of the artifact are limited by the researcher's knowledge, the general limitations by the context, and the approach's feasibility. The researcher needs for this step substantial knowledge of the problem scope and the efficacy of present solutions. (Peffers et al., 2007)

Activity 3: Design and Development

This activity embodies the core of the design science research process as its outcome is the design artifact. Design research artifacts can have different forms, for example, software, models, or appropriate knowledge. Please refer to the first guideline by Hevner et al. (2004) for more details. The researcher needs for this step to be knowledgeable about techniques and theory on how to design and develop the selected type of artifact. (Peffers et al., 2007)

Activity 4: Demonstration

Applying or executing the artifact in one or more problem contexts shows how to apply and utilize the research solution. Methods of demonstration range from experiments over case studies to simulations. The researcher needs for this step practical knowledge of the artifacts usage and a deep understanding of the problem. (Peffers et al., 2007)

Activity 5: Evaluation

While the demonstration shows how to apply the artifact, the evaluation step shows how well it works concerning the identified problem. Accurate observation of the demonstration allows the researcher to gather data for the evaluation. Comparing the defined objectives in Activity 2 with the results from the demonstration in Activity 4 is one of many forms of how a researcher can evaluate an artifact. Qualitative and quantitative performance metrics are other ways for evaluation. In general, any empirical or logical proof can be used as an evaluation method. Depending on how well the evaluation result corresponds to the objectives' accomplishments, the researcher can decide if

the design and development activity should be revisited or the efficacy is satisfactory for the available resources. The researcher needs for this step knowledge about methods of evaluation, a deep understanding of the problem, and adequate means to gain meaningful data to evaluate. (Peffers et al., 2007)

Activity 6: Communication

The audiences of interest will be made aware of the findings of the research. Interested entities can be executives of companies and governmental institutions, affected operators who are targeted to apply the solution, or other researchers and the related scientific community. Topics of the communicated contents are the problem itself and its relevancy for specific parties, the solution in the form of an artifact, and how it represents a novel approach to solve the underlying problem. Other important matters to communicate are the artifact's efficacy to solve the indicated issues, how to deploy and utilize the artifact, and how the artifact was designed. The researcher needs for this step knowledge about the related scientific field and its principles and customs. (Peffers et al., 2007)

Taking the problem at hand, its affected parties, and the chosen solution approach into account, would hint to a good fit for a design science research project (DSR). The issues raised in the introduction section mainly concern managerial and operational audiences. The intended solution is focused on designing and applying an enhancement of current practices rather than on evaluating hypotheses and providing potential explanations to a scientific community. Design science seems to be a better fit than behavioral science as the result of this thesis' research is an instantiation in the form of applicable software which changes the phenomenon rather than describes it. The designing and development of the artifact was realized in the sense of an evolutionary prototyping software engineering method described and categorized as design-oriented by Wilde and Hess (2007). However, there are motives that lead to disagreement of classifying this thesis' result as a design artifact rather than as a product of professional software engineering. The next section will go into

more detail whether the developed software construct can be seen as a design artifact in the sense of design science.

2.3. Software Engineering Versus Design Science

Ross et al. (1975) define software engineering as applying engineering methods and tools to produce software. Software is, therefore, the realization or instantiation of concepts, models, and architectural designs. Hevner et al. (2004) state in their first guideline of design science research the necessity of a design artifact as a result. March and Smith (1995), Hevner et al. (2004), and Gregor and Hevner (2013) explicitly define instantiations as possible artifacts suited for DSR. However, the author of this thesis defines his research outcome as an IT artifact in the sense of software engineering and does not try to hold up to the requirements of a design artifact with respect to DSR. This section will explain in more detail the author's opinion why his thesis should rather be considered as a software research and development project than a design science research project.

Hevner et al. (2004) define *constructs*, *models*, *methods*, and *instantiations* — based on the earlier definition of design artifacts by March and Smith (1995) — as artifacts in the sense of DSR. They refer to instantiations as: "implemented and prototype systems" which fits the core idea of the software library implemented within this thesis. One can argue that the artifact itself is a concept or a theory which is only demonstrated by an instantiation. March and Smith (1995) see an instantiation as the implementation of an artifact within a relevant context. However, March and Smith (1995) also explicate that:

> "... an instantiation may actually precede the complete articulation of its underlying constructs, models, and methods. That is, an IT system may be instantiated out of necessity, using intuition and experience. Only as it is studied and used are we able to formalize the constructs, models, and methods on which it is based."

Gregor and Hevner (2013) later clarify their position on instantiations as proper design science artifacts as follows:

> "... we would still include the artifact or situated implementation (Level 1) as a knowledge contribution, even in the absence of abstraction or theorizing about its design principles or architecture because the artifact can be a research contribution in itself. Demonstration of a novel artifact can be a research contribution that embodies design ideas and theories yet to be articulated, formalized, and fully understood."

Higher levels of knowledge contribution are, for comparison, made up of Level 2 as "nascent design theories" and Level 3 as "well-developed design theories". (Gregor & Hevner, 2013)

According to these positions on the definition of design artifacts, one could argue that the software implementation of this thesis suffices the first guideline by Hevner et al. (2004). It can be seen as a Level 1 contribution to the design science knowledge base and would therefore be a valid design artifact. Alter (2006) even wrote a paper about the question of what an IT artifact is, called "Work Systems and IT Artifacts - Does the Definition Matter?" He concludes that the interpretation of what an IT artifact is does indeed make a significant difference for the researcher and for the audience.

Design science is still a new research paradigm and is therefore still affected by discussions about definitions and boundaries to other academic disciplines. Offermann et al. (2010) list several perspectives and interpretations of what can be identified as a DSR result, based on the opinions of numerous authors of relevant papers. Besides the question if instantiations are design artifacts, their paper emphasized the challenge of certain quality aspects of artifacts in general. The primary metric with respect to the level of quality for artifacts is the contribution to the DSR knowledge base through the artifact itself.

The *DSR Knowledge Contribution Framework* by Gregor and Hevner (2013) categorizes the effects of an artifact on design science knowledge by two dimensions. Depending on the maturity level of the

application domain and the solution itself, a research falls into a distinct quadrant. The area situated in the higher end of both scales represents a *routine design* with no major additions to the knowledge in the DSR space as it mainly lacks novelty. Gregor and Hevner (2013) emphasize the importance of differentiating between design artifacts and products of conventional software engineering that would fall in the *routine design* category. They see it crucial "that high-quality professional design or commercial system building be clearly distinguished from DSR." Design research has to extend the descriptive (Omega or Ω) knowledge base which explains phenomena or append to the prescriptive (Lambda or Λ) knowledge base which shows how to create artifacts. (Gregor & Hevner, 2013)

> "The key differentiator between professional design and DSR is the clear identification of contributions to the Ω and Λ knowledge bases in DSR and the communication of these contributions to the stakeholder communities." (Gregor & Hevner, 2013)

The author of this thesis shares the understanding of the importance of knowledge contribution as characteristic of DSR by Gregor and Hevner (2013) and sees the result of this thesis in the *routine design* quadrant. The produced artifact is embedded in a known problem domain and applies common solutions. The innovative factor is not pronounced enough to constitute a significant addition to the knowledge base of DSR. The rigor required for DSR is also challenging to maintain with the limited resources of a single composer within the scope of a master thesis. However, the proposed process model of the DSRM by Peffers et al. (2007) fits also well for an iterative software engineering project and can therefore still be applied, even though it is technically not a design science research project.

The next sections introduce the reader to the author's approach of evolutionary prototyping, which was augmented by activities and ideas proposed by Peffers et al. (2007) in their DSRM.

2.4. Applied Methodology

The methodology chosen for this software research and development project is a combination of evolutionary prototyping and the design science research methodology (DSRM) by Peffers et al. (2007). This section describes how DSRM was applied to this project (see Figure 2.3 for an outline).

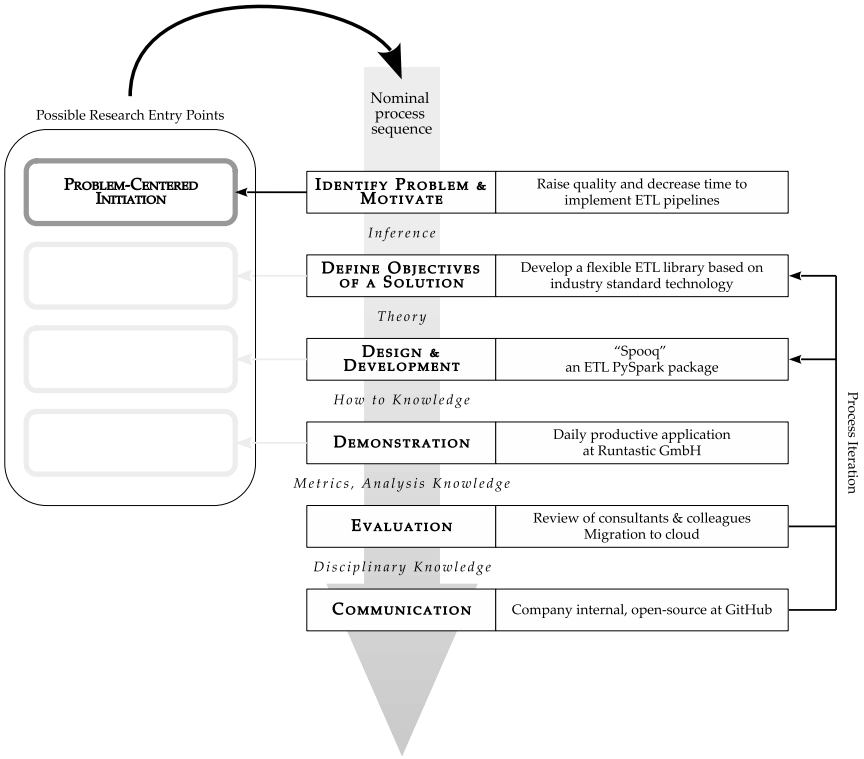


Figure 2.3.: Applied Design Science Research Methodology (DSRM) Process Model - Based on Figures Provided by Peffers et al. (2007)

2.4.1. Problem Identification and Motivation

The first step for problem induced research is, according to the DSRM by Peffers et al. (2007), to identify the problem and its effects on

different groups. The author discovered the problems and identified the issues effected in practice at his daily work as a data engineer. Options to solve the problematic situation were found through personal research and interchange of ideas with persons internal and external to the author's company.

The author was made aware of the general problem field through his work at Runtastic GmbH. He has been working for more than five years as a data engineer, where his primary responsibilities have been to ingest data into a data lake, transform it appropriately, and persist it to a data warehouse. Multiple ETL process implementations uncovered the inherent complexity of designing and constructing data pipelines within data lakes.

Discussions with affected persons gave an impression of the severity regarding lost value due to complexity-initiated issues. Resource planning, unmet expectations of stakeholders, and the average time taken to finish affected projects have affirmed the consequences of the matter.

Continuous discussions with colleagues and management helped to define the issues more clearly. Talks and assistance from third parties like commercial support, external consultants, and other data-driven companies have confined the scope of the problem terrain. Research within the scientific literature, existing open-source projects, and among other people in the community of interest has provided an overview of possibilities to solve or mitigate the issues.

2.4.2. Objectives for a Solution

The initiation of the examined research topic is problem-centered and allowed, therefore, to infer objectives directly from the problem definition. The motivation from different stakeholders for a solution defined a general scope of effects that the constructed artifact should accomplish. The context and environment of the problem provided requirements and limitations which the research's result has to adhere

to. The type of the artifact dictated involved fields of knowledge, each presenting different principles to comply with.

Problem-Specific Objectives
Objectives and constraints were defined for the problem sphere. Key indicators to evaluate the efficacy of the solution with respect to the identified problem were determined. Different scenarios with heterogeneous environments were defined to showcase the generalization of application.

Principles of Data Engineering
Principles of data engineering were used to derive metrics for and methods of evaluation. The purpose of the artifact and its use are mainly related to the field of data engineering and its practitioners.

Principles of Software Development
The problem confinement and objective definition led to the implementation of a software library called *Spooq*. Software development provided additional fundamentals by which the research's software instantiation should be judged. The resulting artifact is a software library which therefore has to adhere to software development principles.

2.4.3. Design and Development

The conclusion from confining the problem and defining the objectives was to create a software library. The purpose of this artifact was to be directly applicable, as a basis for custom implementations, or as applied guidelines for other problem solutions. The development of this software was of agile nature, which conforms with design science principles in general and iterative prototyping as a development method in particular.

A first prototype was created in 2017 to help with a very narrow use case. The implementation — plainly named `Job Helper` — was

merely a sole Python class to be used to incrementally ingest a single entity type from a MySQL database into Runtastic's data lake via JDBC, which was previously done with the help of Apache Sqoop. The creation of this module was beneficial in comparison to the previous application for specific reasons. Spark 1.6 was the clear choice for the application's base framework due to practical reasons within the environment and also in foresight to general usability and portability. Re-implementing an existing use case helped in evaluating the results of the application in comparison to the former implementation. The utilization and demonstration of the new artifact produced useful insights to be evaluated against the preceding solution and the projected intents. Uncovered bugs, improvable performance of predefined objectives, constructive criticism by colleagues, and especially extensions of the underlying use case set the development back in the life cycle to the first activity described by Peffers et al. (2007) to update the problem definition and resume the successive steps.

After several iterations of the module's life cycle, it became clear that the immanent architecture at that time was heavily limiting its extensibility, quality, and general applicability for other cases. July of 2018 was the actual start of the current artifact, which is subject to this thesis. *Spooq* — a play on words with Apache Spark and Apache Sqoop — was the new name, reflecting a fundamental restructuring of the code and widening the application's purpose. Learning from previous iteration loops profoundly helped to understand the problems from a more general point of view, both in content-related and temporal dimensions.

External influences forced the artifact to evolve besides purpose and evaluative reasons. Spark 2.0 was released in 2016, which added beneficial changes in functionality, quality, and performance compared to Spark 1.6. Bugfixes made some workarounds of *Spooq* redundant, but breaking changes of this major update required to adapt the syntax of the Python language binding. *Spooq 2.0* was released at the beginning of 2020 to support Spark 2. The release of Spark 3.0 — still in preview as of beginning 2020 — and the end for official Python 2 support by the Python Software Foundation will make it necessary for further iterations of the artifact. The future will provide new challenges and

opportunities, which will result in refined and enhanced versions of *Spooq* to remain relevant and usable. (Apache Software Foundation, 2020; Python Software Foundation, 2020)

2.4.4. Demonstration

The presentation of the artifact was continuous and iterative with similar loops as for the design and development stage.

Spooq started with solving a very specific use case, and further adaptations were often made to support new cases of applications. This led to constant demonstration after each iteration loop by applying the software library in the daily ETL processing at Runtastic.

Tests
: Unit tests are a natural way of showcasing the functionality of *Spooq*. For each new feature, multiple automatic tests were made to ensure high quality and discover bugs early on. The entirety of *Spooq's* test suite shows what the software can do and that it works — at least in the test-specific environment.

Code Reviews
: Every change in code had to be reviewed by at least another data engineer before it was allowed to be used productively. This helped to uncover errors, ambiguities, and other issues. Comments on the code gave essential insights on which the artifact could be improved.

Deployment and Application
: Deploying and utilizing *Spooq* was the primary way of demonstration. After the library proved its usefulness in a unique testing environment, it was applied to the daily ETL pipeline for productive data.

Demonstration was mainly dependent on and triggered by added functionality, which happened cyclically. This demanded for cyclic demonstration as well.

2.4.5. Evaluation

Evaluating the artifact was a permanent part of the iterative design process. The results were evaluated within each development cycle as the objectives also changed frequently due to the agile approach.

Version 2.0 of the artifact represents a milestone of *Spooq's* development and is the state of subject for this thesis. Therefore, the evaluation process was focused on the latest iteration's objectives and inferred assessment criteria. General efficacy was evaluated, confirming with the objectives from step two. Measurements inferred from the principles of data engineering and software development were checked. Qualitative judgment was gained by consulting the interviewees from step two again after demonstrating the software to them. Those interviews showed if people see an improvement of the problematic situation through using *Spooq*. A test case of migrating *Spooq* to an on-premises Hadoop distribution and to a cloud-based Spark environment attested its general applicability. The evaluation part of this thesis will mainly focus on the last version of *Spooq*, based on its most current objectives and criteria.

2.4.6. Communication

The first group of persons to communicate the end result were the colleagues of the author. Current data engineers at Runtastic can use the library themselves to speed up their ETL pipeline development and rely on thoroughly tested code. New and future data engineers were and will be introduced to *Spooq* to facilitate their entry to ETL processing within data lakes.

The direct management for data engineering at Runtastic was shown the result and explained the implications of using such an artifact, supported by the evaluation conclusions. External persons who have relations to Runtastic with respect to data engineering were also made aware of the new library.

The author thought about different channels and media to communicate the thesis' result. Runtastic provides a tech blog that could be used as a communication channel for interested individuals. Other means of expression is to mention it to interested persons of other companies directly.

The primary way of communication will be through open-sourcing the code to enable other engineers, executives, and researchers to use, modify, or just get inspired by *Spooq*.

The next part will introduce the reader to the results of the activities of the applied methodology.

Part II.

Results

3. Problem Identification, Motivation, and Objectives

Section 3.1 explains the problem at hand and how the author became aware of it. The relevancy to a company and the applicability to other businesses is described further on. Section 3.2 continues with appropriate objectives and evaluation criteria for a potential solution to aforementioned challenges.

3.1. Identification of the Problem and Motivation for the Solution

The author has been working as a data engineer at Runtastic GmbH in Upper Austria, where he has had extensive contact with many different forms and sizes of data. Runtastic provides multiple mobile applications with which end users can track and save health and fitness-related data and metrics. As of 2019, the company had a staff of over 240 employees with more than 35 different nationalities, distributed over three offices in Linz, Salzburg, and Vienna. Over 300 million users downloaded Runtastic's mobile applications over the last ten years and gave them an average rating of 4.4 out of five stars. Adidas bought the company in 2015, which allowed for well recognized worldwide campaigns, like *Run for the Oceans*, where over two million participants ran and recorded 12.6 million kilometers

in just a few days. Almost 150 million customers have an account and share their data with Runtastic. (Adidas GmbH, 2016; runtastic GmbH, 2019a, 2019b)

At Runtastic, the author of this thesis has been mainly responsible for developing and maintaining ETL (Extract-Transform-Load) data pipelines within a data lake. ETL provides an essential part in transitioning data from different source systems into a unified data warehouse or data lake. When implementing and maintaining a data warehouse, data engineers spend 70 percent of their resources for ETL activities, according to Kimball and Caserta (2004). The extraction process of ETL obtains relevant data from a source system in either a static way — all data is extracted at once — or incrementally, where only deltas are extracted. The extracted data gets converted into a usable form during the transform activity. This includes cleaning, augmenting, and pivoting, among other steps. Lastly, the loading step is responsible for storing the transformed data in a query-able format, in an accessible place, for the right consumers. ETL pipelines cover the physical and logical journey of data from its origin to a target area used for actionable insights. Section 4.1.1 will go into more detail on this topic. (Kimball & Ross, 2013)

The data of Runtastic's users is stored and processed for analytical usage in anonymized form in a data lake, based on an Apache Hadoop distribution by Cloudera. The data software stack consists mainly of open-source software like Apache HDFS, Apache Hive, and Apache Spark. User expertise, community support, and knowledge is, therefore, not limited to the Cloudera ecosystem. Companies, engineers, and managers, who are using products based on one of those open-source technologies, are assumingly affected by the same problems, and possible solutions are potentially applicable for them as well. Data processing related problems, identified within Runtastic, are relevant to other companies with similar environments and use cases, following the assumption of a shared software stack. (Cloudera, Inc, 2015; Semlinger & Litzel, 2016)

The challenges and required skills of data engineers have changed over time due to the increasing volume of data. According to Anderson (2016a, 2016b), there is a significant increase in subjective complexity for big data projects in comparison to other emerging software technologies like mobile or cloud. Data engineers who are responsible for big data pipelines have to work in a more structured and architectural way, due to the involvement of multiple services and different distributed systems, which are common in this area.

With the advent of large and distributed datasets, a new type of data engineer has been emerging. These so-called "big data engineers" are less focused on declarative programming languages, like SQL, than their general counterparts. Their main tools are procedural programming languages due to the nature of open-source software primarily used in data lakes. As discussed later in Section 3.2, code-based ETL development provides many advantages over using GUI (graphical user interface) based applications. A main disadvantage is, however, the added complexity for even simple workloads. "Big data engineers" gain the freedom of creativity but lose the abstraction of domain-specific languages. (Anderson, 2016a, 2018; Cloudera, Inc, 2015; Santos et al., 2017)

The author became aware of the substantial complexity of ETL processes within a data lake after he had implemented multiple data ingestion pipelines for different entity types from various sources. At this stage, no dimensional modeling techniques are applied in the data lake at Runtastic, which is why facts (e.g., a run by a user) and dimensions (e.g., the attributes of a user) are treated equally and are further on referred to as entity types. Training, knowledge transfer, and support by the software vendors made clear that difficulty and effort to implement ETL applications in a data lake was caused by the software frameworks themselves with respect to the use cases rather than by naive implementation and usage of ETL systems. Interchanging of ideas with managers and big data engineers from different companies all over Europe showed that Runtastic's use cases and infrastructure are rather common and not specific to Runtastic's data lake.

Further discussions at conferences and events with other companies determined that the Apache Hadoop-based software stack represented the reference implementation of data lakes at that time. A significant part of the data-driven industry shares the same software environment, form and size of data, and use cases for utilizing data. Other businesses experience the same challenges of increased complexity and decreased efficiency for building data pipelines within data lakes. (Pasupuleti & Purra, 2015)

Conversations with Christoph Ferrari, the longtime head of Data Engineering and Data Science at Runtastic, affirmed that the lack of standardized support for ETL procedures has severe implications for the company. Development by the data engineers takes a long time due to missing guidance and backing of a reusable and configurable methodological framework. Some critical business decisions can not be taken with the necessary information in time or analysis, and reports on specific entity types will not be achieved at all because it would take too many resources.

No standard methodology with best practices for extraction, transformation, and loading of data within data lakes has been established. The openness of data lakes, in combination with the independence of open-source software, presents data engineers with a sheer endless number of options, which consequently hinders standardization. (Santos et al., 2017)
At Runtastic, ETL pipelines were written in a multitude of languages and corresponding computation engines. There were scripts written in SQL, HQL (Hive Query Language), Impala SQL, Bash, Python, Java, Scala, or PySpark, depending on the use case and the skill set of the executing data engineer. Most of these scripts were not tested nor documented due to their single-use application and technological heterogeneity. Due to the schema-on-read principle and the variety of software tools to choose from, the openness of data lakes results ultimately into a limitation.

ETL processes often share somewhat similar logical steps for many data pipelines within a company. It is, therefore, possible to use best

practices from software development and abstract common processing methods to be used by multiple ETL processes. Combining a shared codebase with the utilization of metadata enables data engineers to decrease their coding effort perceptively. Creating, adapting, and maintaining relevant information about available datasets, their interconnections, and requirements for potential use cases forms the essential basis to operationalize metadata. Business rules can be applied upon this knowledge to allow automating ETL processes by selecting the right actions and inferring parameters, which are applied to the chosen components. If done in a modular and extensible way, exploiting metadata avoids code duplication and prevents the re-invention of the wheel for every new type of entity to process.

Talks with multiple external consultants confirmed that an ETL framework which is reusable and configurable would be helpful also for other companies coping with the same problems. An abstraction of the processing steps to the level of business logic could substantially decrease the complexity of data lake-based ETL procedures and therefore speed up implementation time.

The identified problems of data processing within data lakes can be summarized to the following points:

Increased complexity due to the high variety of data
: Virtually endless types of structured (e.g., CSV files), semi-structured (e.g., XML files) and unstructured (e.g., image files) data hinders standardization. The schema-on-read paradigm pushes the complexity of structuring the data from the source into the data lake, as it is usually imported in its rawest form possible.

Increased complexity due to the software stack
: Data lakes make mainly use of open-source tools which come in all variations and maturities for numerous use cases. Developing data pipelines with those tools requires writing programs instead of defining the business logic in the majority of cases.

No established standards to provide conformity
Facilitating uniformity of pipelines and abiding standards is difficult as there are too many distinct software frameworks which are overlapping but still do not cover the complete set of needed functionality.

Low quality of data pipelines
High-grade data and stable ETL processes are based on proper testing and documentation of the code. Lacking standards, alternating frameworks, and increased development time put significant constraints on those activities.

Missed business value due to long development times
The elevated complexity leads to a rise in development time and consequently, to delayed or even missed business opportunities.

3.2. Objectives for the Solution

This section identifies the objectives for the resulting artifact of this thesis. Section 3.2.1 derives objectives from the problem space itself. Big data engineers are the main actors to apply and use the artifact which allows Section 3.2.2 to derive further objectives from the principles of data engineering. Principles of software engineering are examined as well due to the software-based nature of the artifact. They are examined in the same section, as they mostly overlap with data engineering. Section 3.2.3 categorizes the outlined objectives from the two previous sections and formulates them as evaluation criteria.

3.2.1. Problem-Specific Objectives

The identified problems are embedded in environments that use mainly open-source software to operate data lakes. The proposed

solution should, therefore, be applicable in such situations. A code-focused implementation is favored in comparison to GUI-driven software. This should, however, not lead to an application which is cumbersome to use by data engineers or data scientist. The resulting artifact of this thesis should provide functionalities for semi-automated batch-processes and ad hoc use cases.

Code-Based Interface

"Big data engineers," working with data lakes, are used to write pipelines and scripts in procedural languages, which calls for a solution with a code-based interface. Section 4.1.1.1 will go into more detail on the additional benefits of a code-driven solution in contrast to a graphically-driven implementation.

Functionality

General ETL processes should be possible with the proposed solution. This includes extracting data from a source, transforming it into a usable dataset, and lastly, loading the output into a database or other destinations. This satisfies the use case of daily batch jobs which decode raw data stored in the data lake and persist it in a usable and accessible form.

Another use case to serve is providing a way to use the library for ad hoc queries accessing raw data. Data Scientists often need additional information to what is stored in data warehouses or other databases. The application should, therefore, enable users to easily load raw data on a request basis to be used for various analyses or development. This is sometimes also called ELT (Extract, Load, and Transform) for data lakes.

Ease of Use

The software library should enable data engineers, data scientists, and other operators in a data lake to construct and execute data processing pipelines in a non-verbose and easy-to-apply manner. The creation of data pipelines should be lifted from the level of implementing processing code to the level of defining business logic.

Broad Applicability

The problem-centered initiation of this thesis leads, in the first place, to a solution that solves the problems in the originating environment. The library should, however, be applicable in comparable areas as well. Therefore, it should be potentially utilizable in other data lakes, besides Runtastic, providing a typical data processing software stack.

3.2.2. Goals and Principles of Data Engineering and Software Development

This thesis addresses especially code-focused "big data engineers" who are used to write ETL pipelines with procedural programming languages. Principles of software engineering were, therefore, also considered, next to data engineering. This section combines the rationales of both disciplines as they overlap in most parts.

The proposed implementation, called *Spooq*, should be built with the ability to modify and extend its functionality with reasonable effort. Frameworks and software used by *Spooq* should be well-known and available in common data lakes to reduce friction of operation and implementation. Scalability is essential, as processing big datasets is one of the main requirements. The reliability and understandability of the application should be kept in focus to enable productive deployment and operation.

Modifiability

Ross et al. (1975) name modifiability as one of the primary goals of software engineering. Adaptations to the code should produce only the desired outcome without interfering with already existing functionality. Confined alterations are of significant importance as software — especially open-source software — almost never "finishes" and will be modified as long as it is used. Modifiability also relates to the ability to change the context of the program rather than the code itself. This can

be a software's universal applicability on different computers, operating systems, or cloud providers. (Ross et al., 1975)

Evolvability is one major aspect of the ability to maintain data-intensive applications. Requirements change frequently. The constant need to adapt software requires an architecture with decoupled components to keep unintentional effects to a minimum. Well-defined abstractions help other engineers to understand the application, which consequently makes it easier for them to modify it with confidence. Simplicity often directly translates to modifiability. (Kleppmann, 2017)

Efficiency

To use the least amount of computational resources to achieve the biggest effect possible is a desirable goal of software development. However, Ross et al. (1975) argue that the subject of efficiency should always be considered in relation to other — potentially more important — aspects of a program. Although great inefficiency should not be tolerated, tweaking for the last bit of performance is neither necessary nor advisable. (Ross et al., 1975)

In contrast, for the data-intensive applications which process big data, efficiency plays a major role. Implementations which work well for a few thousand records can take days to process Petabytes of data. This quality is usually referred to as throughput in data processing use cases. Response time and latency, on the other hand, are mostly negligible for ETL processes as those are generally non-time-critical and often batch-oriented. For an application to be suitable for big amounts of data, good scalability (less or equal to linear complexity) is of utmost importance. (Kleppmann, 2017)

Reliability

Software to be considered reliable must be stable and predictable. Errors and mistakes have to be uncovered and corrected in the design and development phases. In addition, programs should incorporate logic to handle unpredictable effects and corrupt input data at runtime. (Ross et al., 1975)

> Applications whose purpose is to process large amounts of data have special challenges that make reliability difficult. To be able to handle data bigger than the memory of a single server, parallel computing often serves as the basis for big data frameworks. This paradigm combines multiple nodes of a cluster to work simultaneously on a shared set of data. Distributed computation entails the risk of hardware or network failure, which has to be dealt with. Data-intensive applications are commonly designed in a fault-tolerant way by explicitly anticipating problems and providing redundancy for reconstruction. (Kleppmann, 2017)

Understandability

> Computer programs are profoundly complex entities that can not be understood by humans in their totality if they would not make use of abstractions. Strict structures and well-designed architectural concepts greatly help to decrease the perceived complexity of applications. The lesser complexity a software exhibits to a developer or user, the easier it is for them to comprehend the software adequately. Simplicity — the antipode of complexity — directly relates to understandability. (Ross et al., 1975)

> Moseley and Marks (2006) describe that unnecessary logic, which does not originate from the problem itself but the implementation of the solution, increases the intricacy even further, what they call *accidental complexity*. Kleppmann (2017) argues that eliminating unnecessary logic, which does not help for the offered capabilities, can make an implementation simpler without removing any functionality.

3.2.3. Evaluation Criteria

Applying the goals and principles, mentioned in the previous sections, results in criteria which will be used to evaluate the produced solution of this thesis. This section formulates and lists the criteria the author came up with to validate *Spooq's* approach concerning the identified problems. The criteria are ordered and numbered by their main

category (Roman numerals), sub-category (Arabic numerals), and the criteria themselves (Arabic numerals). The abbreviation "EC" stands for "evaluation criterion" and will be used in the remaining text for brevity reasons.

I Providing ETL Functionality for Big Data

This category is split into two parts. The necessary functionality for ETL processes is derived from the problem-specific objectives at Section 3.2.1. The ability to work with big data relates to efficiency.

I.1 Functionality

In order to evaluate the functionality of *Spooq*, at least one example of the following components should be implemented using *Spooq* and each should be validated to be operational:

I.1.1 One Extractor

It will be demonstrated that *Spooq* is able to extract data from a source.

I.1.2 One Transformer

It will be demonstrated that *Spooq* is able to apply transformations on data.

I.1.3 One Loader

It will be demonstrated that *Spooq* is able to load data into a target system.

I.1.4 One Pipeline

It will be demonstrated that *Spooq* supports combining an extractor, transformers, and a loader into a single pipeline object.

I.2 Scalabiltity

The scaling capabilities of *Spooq* will be evaluated by an in-depth discussion of *Spooq's* support for following characteristics:

I.2.1 Parallel computing

It will be discussed that *Spooq* supports parallel computing.

I.2.2 Horizontal scaling

It will be discussed that *Spooq* supports horizontal scaling.

I.2.3 Cloud compatibility

It will be discussed that *Spooq* can be utilized in cloud-based deployment scenarios.

II Decrease Complexity of Data Pipelines

The ease of use for generating data pipelines with *Spooq* covers the problem-specific objective at Section 3.2.1 by reducing the application's complexity.

II.1 Parameterizable

Spooq should be fully configurable with parameters. No additional processing methods should be needed for a data pipeline. The following exemplary use cases will be used to evaluate *Spooq's* ability to configure pipelines via parameters:

II.1.1 Daily Batch-Processing

It will be demonstrated that a batch-based ETL process can be configured without the need to access under-lying methods of *Spooq's* base framework. This use case will include extraction, cleaning, filtering, restructuring, and loading of user data.

II.1.2 Ad Hoc Data Preparation

It will be demonstrated that an ad hoc data preparation pipeline can be configured without the need to access under-lying methods of *Spooq's* base framework. This use case will include extraction, restructuring, and cleaning of the data.

II.2 Semi-Automatic Configuration by Reasoning

Configuration of *Spooq* applications should be supported in a semi-automated way with only a few input attributes required. Pipelines should be configurable without explicit parameters if

context variables and relevant metadata are provided. The following two use cases will be used to evaluate this functionality:

II.2.1 Daily Batch-Processing

It will be demonstrated that a batch-processing ETL pipeline can be automatically configured on basis of less than or equal to five input parameters. The use case from EC II.1.1 will be re-used for this demonstration.

II.2.2 Ad Hoc Data Preparation

It will be demonstrated that an ad hoc data preparation pipeline can be automatically configured on basis of less than or equal to five input parameters. The use case from EC II.1.2 will be re-used for this demonstration.

III Conform with Standards and Best Practices

Sections 3.2.1 and 3.2.2 provide standards and best practices for data and software engineering. This evaluation category lists three sub-categories which are further broken down and formulated as evaluation criteria.

III.1 Code-Focus

Data lake-based ETL pipelines are often designed and developed in scripts or notebooks, based on procedural programming languages. *Spooq* should take this into account and support code-based development.

III.1.1 Code-Based Interface

It will be discussed that the main interface to *Spooq* is provided via code in a programming language that is common in data engineering and data science.

III.2 Broad Applicability

Spooq should be utilizable in different data lakes environments. A *Spooq* data pipeline will be successfully executed in following environments:

III.2.1 Stand-Alone Spark

It will be shown that a ETL pipeline based on *Spooq* works correctly on local hardware, running in stand-alone Spark mode.

III.2.2 On-Premises Hadoop Cluster

It will be demonstrated that the batch-processing use case from EC II.1.1 works correctly on an on-premises Hadoop-based Cloudera cluster, running against a YARN resource manager.

III.2.3 Cloud-Based Databricks Cluster

It will be demonstrated that the ad hoc data preparation use case from EC II.1.2 works correctly on a cloud-based Databricks workspace.

III.3 Evolvability

The implementation of new data sources should not affect the usage of other transformers or loaders. Additional data sinks should not affect the effects of accompanying extractors or transformers. Newly introduced transformers should be compatible with previous extractors and loaders. The evolvability will be evaluated by demonstrating the necessary effort to implement the following exemplary components:

III.3.1 One Extractor

The necessary code changes for the implementation of an example extractor class will be analyzed.

III.3.2 One Transformer

The necessary code changes for the implementation of an example transformer class will be analyzed.

III.3.3 One Loader

The necessary code changes for the implementation of an example loader class will be analyzed.

IV Increase Quality of Data Pipelines

As described in Section 3.2.2, tests increase the reliability, whereas documentation can help users to understand how to operate *Spooq*.

IV.1 Testing

Spooq should be designed to support unit testing. The testability will be evaluated by the code coverage of its included test case and by demonstrating the effort to write tests for new components.

IV.1.1 Code-Coverage

It will be demonstrated that at least 75 percent of each implemented extractor, transformer, loader, and pipeline code is covered by unit tests.

IV.1.2 Writing Unit Tests

The necessary code changes for the implementation of unit tests for one extractor, one transformer, and one loader component will be analyzed on basis of the exemplary implementations for EC III.3.1, EC III.3.2, and EC III.3.3, respectively.

IV.2 Documentation

Spooq should be designed to automatically generate most of its documentation from its source code. The documentation should be available through different channels.

IV.2.1 Formats

It will be demonstrated that *Spooq's* documentation is provided as HTML and PDF.

IV.2.2 Documentation by Source Code

The necessary code changes to document one extractor, one transformer, and one loader component will be analyzed on basis of the exemplary implementations for EC III.3.1, EC III.3.2, and EC III.3.3, respectively.

4. Design and Development

The start of this section gives technical basics to understand the design and use cases of *Spooq* better. ETL transformations, the Apache Spark framework, and the expert system *Experta* are covered. The implementation section features more information on the architecture, separate components, and auxiliary services of *Spooq*.

4.1. Technical Basics

This section gives some background on the technical aspects of relevant fields of knowledge for this thesis. Section 4.1.1 starts with a description in more detail about the transformation actions in ETL processes, which is central to this thesis. Apache Spark, which is used as the basis for *Spooq*, is explained in a subsequent part. How to infer information from metadata and business rules is specified in Section 4.1.3, which is needed to realize the semi-automatic configuration functionality of *Spooq* pipelines to decrease the complexity for the executing user.

4.1.1. Transformations in ETL

ETL describes the process of importing meaningful data into a data-centric platform, like a data warehouse or a data lake. Sourcing data from an external origin is the primary purpose of the first ETL step. Following the extraction, transformations establish a compliant

state of the information through various processes. Loading represents the action of persisting the previously transformed dataset to a non-volatile, data storing environment. The transformation part is presented in a more detailed way in this section, as it is undoubtedly the most challenging and complex action in the ETL process.

To transform extracted data into a usable form, generally, two types of processes are applied: syntactic and semantic processes. Syntactic data transformations mainly include the adaptation of data structures without deleting nor generating information. A common example is to flatten hierarchical data to ensure ANSI-ISO SQL standard conformity or to convert different units to a single standardized type, for example converting attributes defined in different systems of measurement to the International System of Units (SI). This also includes simple data type conversions like strings to integers. In contrast to syntactic processes, semantic steps inherently remove, change, and add information. (Wolfgang Bartel, 2013)

As the quality of externally sourced data is usually quite poor (Golfarelli & Rizzi, 2009; Kimball & Ross, 2013), data cleansing becomes an important step in the transformation process. Golfarelli and Rizzi (2009) list following inconsistencies as the most frequent reasons for data quality issues which have to be compensated for in the data cleansing step:

Duplicate data
: Multiple records for a single instance

Inconsistent values that are logically associated
: For instance ZIP codes and addresses

Missing data
: For example missing address of a customer

Unexpected use of fields
: Such as a phone number in the email address field

Impossible or wrong values
: For instance a birth date in the year 3001

Inconsistent formats of a single attribute
: For example a mixture of abbreviations, names and offsets for time zones

Inconsistent values for one individual, logical entity
: Such as different spelling for the same street name or typing mistakes

After ensuring a satisfactory quality of the input, the next step is to reconsolidate of data. Extracted (and cleansed) knowledge is still in its operational source format. Using and joining the information with other datasets makes the conversion into a common format inevitable. Here are some of the most common types of transformations, according to Golfarelli and Rizzi (2009):

Conversion / Normalization
: This step is similar to data cleansing but operates on all data sources. Where the cleansing process ensures conformity among a single dataset, the normalization step assimilates all data to fit a predefined, reconciled schema. This can either be to translate units from the metric to the royal system or all strings to UTF-8. It can be classified as a syntactic process as no information is lost nor generated. (Golfarelli & Rizzi, 2009)

Enrichment
: Enriching data with additional or derived information is a semantic transformation. A typical use case is to derive country information from IP addresses when needed (e.g., weblogs). Another example would be to add data-specific metadata. This could be information from the extraction, features, and statistics of the data itself, or data fingerprints through machine learning. (Goldman, 2017; Golfarelli & Rizzi, 2009)

Separation / Concatenation
: Due to their different purposes, extracted and loaded data often have distinct structures. Sometimes it is needed to separate entities. For example, a record for user subscriptions in the operational database can have previous and current periods as sub-entities attached, whereas subscriptions and periods are

needed to be separated and stored as different entity types. Although, it can also be because of performance reasons that entities have to be concatenated for the target system. (Golfarelli & Rizzi, 2009)

Due to new regulations, the anonymization of the data as early as possible sees increasing importance. The General Data Protection Regulation (GPDR), which is in force since the 25^{th} of May 2018, provides data engineers with new challenges. Personally identifiable information (PII), non-essential for analysis, has to be removed or anonymized in a way that a single person can not be identified from the information provided. (Datenschutzbehörde, 2018; Welch, 2018)

ETL substantially adds value to the data, if done correctly. Kimball and Ross (2013) sum up the benefits of data transformation:

- Assuring quality and confidence in data
- Documenting the data flow and lineage
- Adapting data from different origins to be compatible
- Mapping data to well-defined and usable structures

4.1.1.1. Code-Based Development

Data engineers design, operate, and maintain ETL activities. Declarative languages, procedural program code, and GUI-driven applications help those engineers to reduce the complexity of implementing ETL processes. In the area of data lakes and big data, the tool of choice for developing data transformations is code-based programming.

Probably the single most crucial skill for a "typical data engineer", who is not focused on data lakes, is SQL. Most of his/her work is done via SQL commands or an SQL compliant dialect. Databases management systems are often of relational type, to be queried and maintained via SQL instructions. The ETL processes are, therefore, also frequently implemented as SQL statements, and the results are

stored into a relational data warehouse, which again is queried via SQL. Though ETL tools with graphical user interfaces are prevalent among data engineers, their basis still relies on SQL logic and its benefits and limitations. (Anderson, 2018)

The increasingly common "big data engineer" still uses SQL and relational logic, but to a lesser extent. His/Her used tools will usually be among open-source solutions based on Hadoop, NoSQL databases, or big data frameworks like Apache Spark and MapReduce. This entails that the primary ETL logic and other data pipelines are primarily defined and implemented via programming languages like Python, Java, or R. In contrast to the role of a "typical data engineer," a lot of "big data engineers" come from a software engineering background. (Anderson, 2018)

Kimball and Ross (2013) present advantages of hand-coded ETL processes in comparison to out-of-the-box ETL tools which hold true regardless of big or small data. The following benefits describe the advantages of hand-coded over GUI-driven ETL tools:

Automated tests
: Almost every programming language supports at least one test framework which allows to unit test written code. This can also be automated for continuous integration testing. Testing the complete codebase, even after adapting only a small code portions, leads to improved quality of the code itself and ensures consistent quality of the output. It supports, in addition, the work of test managers. (Kimball & Ross, 2013)

Consistency for auxiliary processes
: Object-oriented architecture allows developers to reuse generic code, which is not unique to any entity type or process. Error reporting, validation, and metadata operations can use the same code for most data pipelines and therefore act consistent across executions and pipelines. (Kimball & Ross, 2013)

Easier to understand
: In contrast to stored procedures of an ETL tool, hand-coded

ETL applications tend to work directly on file basis. Aside from easier testing and more straightforward coding, this direct approach represents a standard in software development and is well understood. Standard application logic enables other programmers to quickly help out if more working power is needed or to maintain software by other engineers. (Kimball & Ross, 2013)

Flexibility

Modern programming languages, with repositories of libraries and plugins, provide virtually endless possibilities. Developers can implement almost anything by themselves, if there is no suitable package available which supports their specific use case. (Kimball & Ross, 2013; Thomsen & Bach Pedersen, 2009)

Direct metadata management

While this can be seen as extraneous effort for many data engineers as this is often already supported by ready-made ETL tools, implementing custom metadata management enables a more direct access to the data. As a result, developer are not locked in on a limited number of metadata systems supported by a specific ETL tool, but can access and interact with every system which provides an API. Another advantage crystallizes when the software infrastructure already has services and applications which support metadata. The ETL application's metadata can be converted for importing and exporting and enables compatibility and reusability between those services. (Kimball & Ross, 2013)

An additional, significant benefit, concerning direct metadata management, is to apply metadata to a business rule engine. ETL processes can be customized, and even fully compiled via parameters if the application is built with that functionality in mind. Developing an application that can fetch metadata from any source, convert it to a supported format by a business rule engine allows inferring relevant parameters.

Graphical ETL tools often store their pipeline definitions as proprietary binary blobs. Utilized logic is potentially defined in database objects, like SQL procedures or triggers, which increases the risk of unnoticed changes. A properly designed, code-focused tool uses mainly text files to persist any logic or settings. This enables to track changes with the help of VCS (version control systems), like Git. Many goals of Git, as stated by Loeliger and McCullough (2012), are directly transferable to code-focused ETL applications. *Maintain Integrity and Trust*, *Enforce Accountability*, *Immutability*, and *Atomic Transactions* represent the most beneficial features for data engineering. With version-controlled code, developers and operators can be sure at all times, what has been deployed, changed, and executed. Code managed by a VCS allows for easier CI/CD (continuous integration / continuous delivery). Loeliger and McCullough (2012)

4.1.2. Apache Spark - A General Engine for Large-Scale Data Processing

Spooq is to be applied in data lakes which utilize mainly open-source software. Apache Spark was chosen as the technical basis as it covers a wide range of functionalities necessary for ETL processes. It is well known, widely used, and commonly provided by data lake software environments.

The Apache Software Foundation describes Spark as "a fast and general engine for large-scale data processing," which focuses on "Speed," "Ease of Use," "Generality," and "Runs Everywhere." (Apache Software Foundation, 2018b)

Current volumes of available and potentially useful data often exceed the technical limits of single computers and servers. To not lose out on those opportunities, alternatives are needed. Cluster computing emerged as an alternative to vertical scaling.

In the beginning, special software was written to solve specific problems in the data science and data engineering realm. Google de-

veloped MapReduce to store and batch process information they gathered and stored from the world wide web. The company also created the frameworks Dremel and Pregel, which are used for interactive SQL queries and iterative graph processing, respectively. Numerous projects, like Impala or Storm, emerged in the Apache Hadoop ecosystem, which were designed to solve a single use case. A problem arises in the necessity to use multiple, disparate software for a single data pipeline. Not only setting up, maintaining, and tuning multiple software stacks is cumbersome, the connection and data transfer between the different programs results in slow, error-prone, and inflexible processes. (Zaharia et al., 2016)

In 2010, a group at the University of California, Berkley, published a paper about a new software project they were working on, which is called Spark. Its design goal was to be a one-size-fits-all engine for data processing (Zaharia et al., 2016).
MapReduce was then a widely used framework for data pipeline processing, which uses conventional, commodity hardware. It provides fault tolerance and an out-of-the-box multi-node cluster computation engine. For many use cases — especially acyclic workflows —- MapReduce works well and provides good performance. For applications that reuse objects and data in an iterative way, severe performance limitations appear. These include interactive interaction with the data like business analyses and data science experimentation. Zaharia et al. (2010) propose in their paper Spark as an alternative to get rid of the aforementioned constraints. (Zaharia et al., 2010)

4.1.2.1. Programming Model

Spark is written in Scala, a statically typed programming language that runs within a JVM (Java Virtual Machine). It uses the high-level concept of drivers and executors, which serve different purposes. A developer writes and executes his/her code exclusively in the driver process, which, in turn, launches multiple executors to work in parallel on different cluster nodes to achieve the desired results. Figure 4.1

showcases this principle by illustrating the basic programming model of a Spark application. All user code (i.e., ETL process instructions) is executed in the driver process. Utilizing a Spark built-in function from the user code triggers an action in the so-called Spark Context which splits the operation into multiple smaller operations which are processed by separate processes, called executors or workers. Those processes can run on the same server, on different nodes in the same cluster, or even on different clusters. The Spark Context collects the outputs of each worker, combines them, and returns the result to the user code if a return value is specified in the utilized Spark function. (Zaharia et al., 2010)

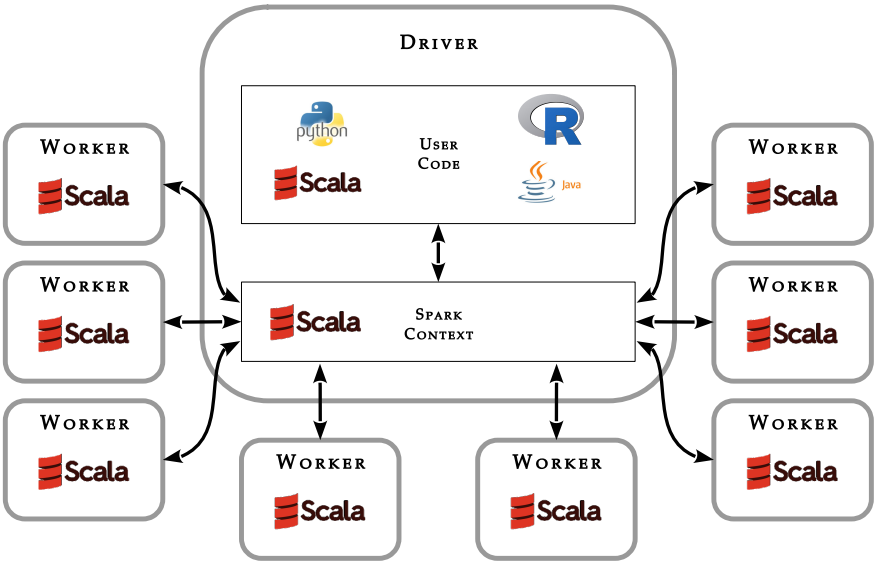


Figure 4.1.: Distributed Programming Model of Spark

Spark offers access to its Scala engine through APIs for Scala, Java, Python, and since 2015 also for R. This allows developers to pass local functions in a functional programming way to the underlying data. (Zaharia et al., 2016)

Resilient Distributed Datasets: The quintessential idea behind Spark is the use of read-only data working sets called RDD (Resilient

Distribution Datasets). They are partitioned across the cluster which means each Spark executor has only access to a subset of the input data. A partition with regards to Spark RDDs is defined as the limited dataset of a single Spark task (sub-operation, local to the worker). The partitioning of the data, sometimes referred to as slicing, is done automatically but can be explicitly defined by the requested number of partitions or by providing a partitioning key. The partitioned datasets can be kept in memory to minimize disk access and vastly improve performance for iterative work. The framework is able to continue and finish jobs, even when nodes or processes crash, like MapReduce. Spark achieves this fault-tolerance through a notion of linage, which stores all previous transformations and copies data from subsequent RDDs. (Zaharia et al., 2010)

Consequently, RDDs do not exist in a physical form but rather as a handle to an execution plan based on physical data stored on HDFS (Hadoop Distributed File System) or other reliable storage systems. Those RDD handles are represented as basic, ephemeral Scala objects, which let the developers easily and seamlessly interact with them. Their lazy evaluation principle lets the developer chain multiple operations together, before an action (collect or reduce operation) triggers the full pipeline processing. This gives Spark the option to optimize the physical execution plan because it has information about all preceding operations. (Zaharia et al., 2010)

RDDs can be created by loading data from a distributed file system by Spark workers in parallel or through the driver which loads data into its process and distributes it to its connected workers. Applying a function — also called transformation — on an RDD creates a new RDD due to their immutability. Only saving or collecting RDDs will not result in another RDDs but in a persisted output or Scala objects within the driver application.

Parallel Operations: At the time of Spark's first public presentation in 2010, RDDs supported three main types of parallel operations: *Reduce*, *Collect*, and *Foreach.*

The Reduce operation uses an associative function to combine multiple partitions and transfers the outcome to the driver process. Counting the number of records falls for example into this category. The count is calculated for each partition in parallel and the results are transferred to the driver process which sums up the received values to come up with the total count. The collecting method is similar to the Reduce operation but skips the aggregating process at worker and driver level. It fetches all data from the partitions defined in the RDD, and transmits it as a Scala collection to the driver application. The Foreach function is probably the most flexible and useful operation type of those three as it allows to define functions that are applied to each element in the dataset. This allows for filtering, mapping, and different kinds of calculations to change data. (Zaharia et al., 2010)

Shared Variables: Spark provides an architectural advantage over MapReduce through shared variables. Sending data, like look-up tables or parameters, to executors, every time a function is applied on a record, results in redundant and potentially avoidable overhead. Spark can distribute those variables at the beginning of the map process to all worker processes, which keep them in memory to access them efficiently. Other custom, distributed data types are accumulators. Their single purpose is to provide a shared data object on which workers can add objects, like numbers for sums. The driver program is the only process that can read the results of an accumulator. Fault-tolerance and easy rebuilding is achieved through their add-only design. (Zaharia et al., 2010)

Higher Level Libraries: Using RDDs as the basic concept for every processing, enabled developers to easily extend the functionality by adding libraries on top of it. The best-known libraries, already included in the official Spark release, are Spark SQL, Spark Streaming, GraphX, and MLlib. (Zaharia et al., 2016)

Spark SQL

Spark SQL provides an interface to query and process data

declaratively. Code generation and cost-based optimization are tuned to increase query performance. The central abstraction for this higher-level functionality is the Spark DataFrame API. A DataFrame is essentially a tabular, type-safe dataset defined in an RDD. (Zaharia et al., 2016)

Spark Streaming
While Spark started as a pure batch processing model, near real-time streaming was later also incorporated. Near real-time because it uses discretized streams which consist of tiny batches of data, for example, data received for every 200 milliseconds. Those batches are regularly synced with the previous batches to combine their states. (Zaharia et al., 2016)

GraphX
The ability to apply graph-based computing with vertices and edges was introduced with the *GraphX* library. The flexible partitioning system of RDDs allows for optimized data partitioning, like vertex partitioning schemata. (Zaharia et al., 2016)

MLlib
As the majority of machine learning algorithms and libraries were not designed to work in a distributed fashion, *MLlib* translates multiple algorithms to be used in a cluster. *MLlib* provides more than 50 common machine learning algorithms, like decision trees and alternating least squares matrix factorization. (Zaharia et al., 2016)

As of 2016, there were over 200 different third-party libraries, developed and used by many different contributors, that customize, extend, or complement Spark. (Zaharia et al., 2016).

4.1.2.2. Application

Spark's applicability as a general computation engine enables developers to implement multiple tasks within a data pipeline through a unified API. The lack of explicit data conversion and transfer between

processing steps eliminates performance bottlenecks and error-prone situations. What smart-phones did for the demand of cameras, mp3 players, and telephones, did Spark for multiple specialized data wrangling software projects. As of 2016, the open-source project Apache Spark had more than 1,000 contributors and was used by organizations like CERN and NASA for scientific research. Companies that use Spark range from science and retail, over biotechnology and banking to social networks and mobile application developers. The largest publicly announced Spark cluster consists of 8,000 nodes used to ingest one Petabyte of data per day. (Zaharia et al., 2016)

Spark is nowadays used for a multitude of different use cases to solve various challenges. The following use cases illustrate the most commonly used means of application.

Batch processing

Typical data warehouse applications utilize the ETL principle where data increments are often loaded in fixed intervals. Apache Spark supports data processing in batches which can be triggered in varying frequencies. Batch processing provides business intelligence services with structured, cleaned, and enriched datasets. Feature engineering and offline training for machine learning applications can also be processed in batches. Noteworthy examples are Yahoo's page personalization, Alibaba's graph mining, and Toyota's text mining of customer feedback. (Zaharia et al., 2016)

Streaming

Real-time decision-making requires immediate data, which is supported by Spark through its *stream processing*. Cisco uses Spark for monitoring its network security and Netflix mines its logs via Spark streaming. Often batch processing is combined with stream processing to combine the best of both worlds. (Chambers & Zaharia, 2018; Zaharia et al., 2016)

Interactive Queries

Data engineers and data scientists run iterative, interactive queries for data exploration. These investigative operations are accompanied by creation-oriented use cases of business

analysts who are generating reports and visualizations, either directly or via a database connection to a BI (business intelligence) tool, like Tableau. (Zaharia et al., 2010; Zaharia et al., 2016)

Scientific Applications

There is a myriad of institutions and companies conducting scientific research with Apache Spark. Biotech companies and organizations use Spark to process huge amounts of genomic data. CERN processes enormous data volumes gathered from experiments and analyses. The neuroscientific platform Thunder at Howard Hughes Medical Institute, Janelia Farm, combines batch, stream, and interactive applications to process brain images and apply machine learning algorithms. (Zaharia et al., 2016)

4.1.2.3. Benchmarks

Even at its early stage in 2010, Spark easily outperformed MapReduce for incremental workloads. Figure 4.2 shows a benchmark for an iterative logistic regression task. Hadoop's MapReduce processing engine beats Spark for the first iteration by 174 seconds to 127 seconds but quickly loses its speed advantage for subsequent iterations. Utilizing cached data brings Spark processing time per iteration down to only 6 seconds, which results in a speed improvement by almost ten times for 30 iterations. A similar benchmark of an alternating least squares job yielded an improvement by a factor of 2.8. (Zaharia et al., 2010)

In 2014, a sorting implementation built on Apache Spark, entered the Daytona GraySort benchmark with astonishing results. It beat the previous implementation on MapReduce by a 200 percent improvement with less than 10 percent of nodes. Apache Spark achieved a sorting rate of 4.27 TB/min, compared to the benchmark of MapReduce in 2013, which achieved a rate of 1.42 TB/min. (Nyberg & Shah, 2018)

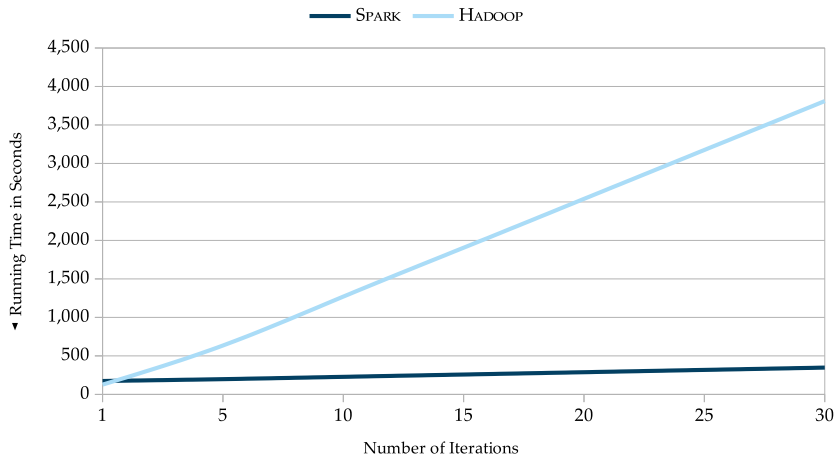


Figure 4.2.: Logistic Regression Performance in Hadoop and Spark - Based on Data Provided by Zaharia et al. (2010)

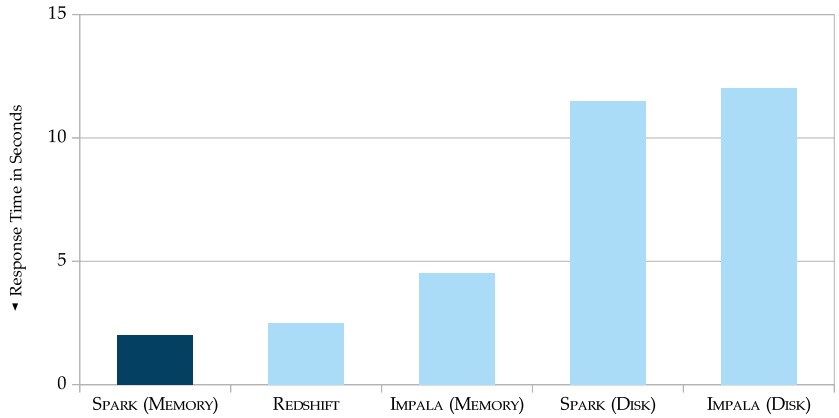


Figure 4.3.: Comparison of Spark Performance Against Widely Used Frameworks Specialized in SQL Querying - Based on Data Provided by Zaharia et al. (2016)

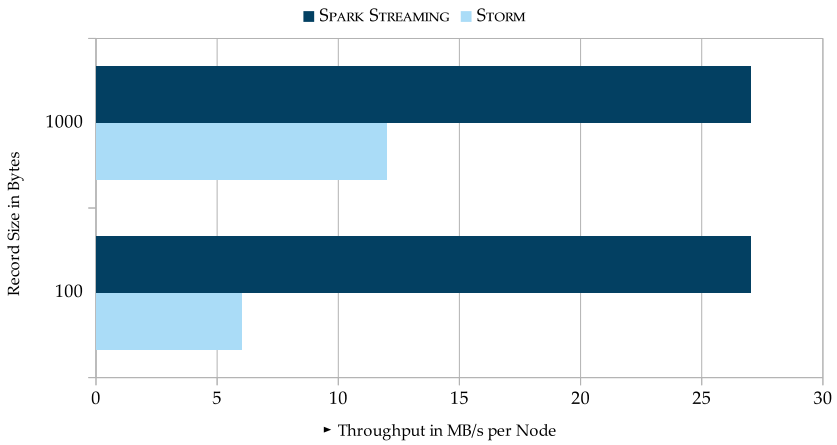


Figure 4.4.: Performance of WordCount Streaming Computing - Based on Data Provided by Zaharia (2016)

Figure 4.3 shows how Spark is competing against frameworks which are specially tailored for SQL queries. Spark achieves to have the lowest response time among all. It also visualizes very clearly that processing on disk is a multitude slower than in memory. Storm, an open-source computing framework specialized in real-time computation, is second to Spark with respect to the throughput of streamed records per second, as illustrated in Figure 4.4.

4.1.2.4. Resource Management

Spark supports three different deployment scenarios where a cluster manager governs the resources and execution of each job. As of 2018, there is also an experimental deployment option with Kubernetes. (Apache Software Foundation, 2018c) This thesis will not concern itself with this alternative any deeper due to its beta status. A fourth option is the local mode, which runs on one single node and is mostly used for testing. All of these options are compatible with either on-premises or cloud solutions. As most companies provide cloud-native Spark services, the developer does not have to take care of the

resource manager and therefore does not need to know details about the resource management of the Spark cluster hierarchy underneath. Although, for on-premises distributions, a good understanding of the underlying structure and management is still crucial. (Chambers & Zaharia, 2018)

Standalone Deployment

The standalone mode provides a light-weight resource management solution to be run on a cluster. The main disadvantage of such a deployment is that the cluster is used exclusively for Spark, with a quick and easy setup as an advantage. (Chambers & Zaharia, 2018)

Deployment on Apache Mesos

Apache Mesos was started by several developers of whom some were also part of the Spark project back in 2009. The first published paper on Mesos describes it as a "platform for sharing commodity clusters between multiple diverse cluster computing frameworks, such as Hadoop and Message Passing Interfaces (MPI)." (Hindman et al., 2011)

A more specific definition is provided by Mesos' website hosted by the Apache Software Foundation which describes it as a distributed systems kernel which "abstracts CPU, memory, storage, and other compute resources away from machines (physical or virtual), enabling fault-tolerant and elastic distributed systems to easily be built and run effectively." (Apache Software Foundation, 2018a)

Mesos' cluster manager is the most heavy-weight out of the supported options. Chambers and Zaharia, 2018 disadvise against the use of Mesos because of its monolithic and hard to handle architecture. Though, if an organization has Mesos already implemented and running, nothing speaks against using it for Spark as well.

Deployment on Apache YARN

This cluster manager is often called Hadoop 2, denoting to

be the direct successor to Hadoop 1, which is often used synonymously for MapReduce. YARN manages job scheduling and cluster resources on Hadoop environments. This makes it tightly coupled with HDFS, which is not always available on cloud-solutions. For on-premises deployments, YARN makes an optimal candidate to be used with Spark and is the commonly recommended solution. As a result of its general approach and compatibility with a large number of different execution engines, configuration can become rather complicated. (Apache Software Foundation, 2018d; Chambers & Zaharia, 2018)

4.1.2.5. Apache YARN

As the execution of Spark jobs and especially its work performance are highly dependent on its resources, understanding the resource management is indispensable. Spark shares a lot of architectural concepts with YARN, which makes it easy to align Spark's application workflow with YARN's provided functionalities. Apache YARN is described here in more detail, as it helps to understand the internal execution and communication logic of Apache Spark.

In 2013, Apache YARN was introduced by a paper by Vavilapalli et al. (2013). They presented their framework as a solution to common problems of then predominant Hadoop MapReduce. With Hadoop, job logic and resource management were tightly coupled and had to be taken care of by the developer on a job level. This led to dirty workarounds and an abuse of the MapReduce framework to compensate for the limitations induced by Hadoop's architecture. The requirements for their software were ambitious. To overcome the limitations and problems of MapReduce, Vavilapalli et al. (2013) list ten requirements:

- **R01** Scalability
- **R02** Multi-tenancy
- **R03** Serviceability
- **R04** Locality Awareness

- **R05** High Cluster Utilization
- **R06** Reliability / Availability
- **R07** Secure and Auditable Operations
- **R08** Support for Programming Model Diversity
- **R09** Flexible Resource Model
- **R10** Backward Compatibility

Fulfilling those requirements enables better horizontal scalability. As Spark's or other engines' distributed job executions are complex workflows, high efficiency, and little downtime for resources is of utmost importance. Generically supporting multiple frameworks opens a path for future execution engines as well. Mesos' offer-based resource allocation leads to a static resource model, whereas YARN's request based model yields a dynamic and flexible resource distribution. (Vavilapalli et al., 2013)

The main principle of YARN's architecture is that it separates resource management and logical execution management. The RM (RecourceManager) service is responsible for tracking, monitoring, and verifying the liveness of resources and of arbitrating between the resources of the whole cluster. Resources or granted leases are subsequently referred to as containers for clarity. A container embodies a logical bundle of resources on a specific node, for example, two GB RAM and one CPU core on node A. The RM holds the sovereignty for granting and revoking containers. To keep track of specific resources, it communicates with the NMs (NodeManagers) of each node, which in turn take care of tracking and managing the life cycle of the containers available on their side. The AM (ApplicationMaster) derives a physical plan for a specific job from the logical outline. It takes into account the available means of processing granted by RM. Fault tolerance is provided by the AM through continuous coordination of the execution, in case of failing tasks or nodes. (Vavilapalli et al., 2013)

Figure 4.5 outlines a basic flow of starting and running a distributed application on a YARN enabled framework, as described by White (2015):

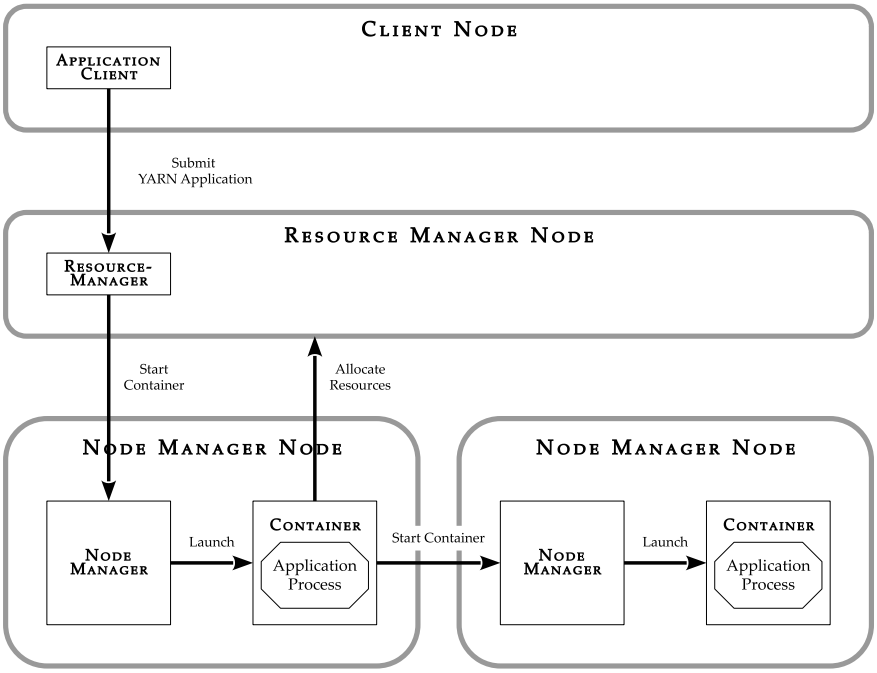


Figure 4.5.: Anatomy of Running a YARN Application - Based on Figures Provided by White (2015)

1. The client initiates the communication with the RM and requests a container to start the AM process within.

2. The RM searches for available containers among its connected NMs. If a free container is available, the RM will launch the AM within the container. The AM will then execute the client's application code.

3. In the case of a YARN aware processing framework, the AM will build a logical execution plan and try to allocate containers from the RM for execution.

4. Depending on the granted resources, the AM will build a physical plan based on its logical plan and spawn new containers to start computation in a distributed manner.

ResourceManager: The RM exposes two public interfaces for orchestrating applications running against YARN. The first one is to submit an application, and the second one is for the AM to request new containers dynamically. Another interface is used only internally to communicate between the RM and the NMs for cluster monitoring and resource access management. (Vavilapalli et al., 2013)

The AM sends the RM a resource request containing the number of needed containers (for example 50 containers), resources per container (for example four GB RAM and two CPUs), locality preferences (to process data on the node where the data is stored), and priority within the application's execution plan. The RM is aware of available resources and its parameters because of the frequent, heartbeat-based communication with the NMs. It tries to fulfill the AM's request as well as possible and provides the AM with delegation tokens to acquire the containers directly from the NMs. In case of fair scheduling, the RM can also revoke already granted resources if a job with higher priority is requesting containers. (Vavilapalli et al., 2013)

It is essential to point out what the RM is *not* responsible for. Coordinating the application itself or supporting fault-tolerance is not part of the RM's tasks. Furthermore, it is not in charge of monitoring and reporting the life cycle of the application. (Vavilapalli et al., 2013)

ApplicationManager: Applications running against YARN can be contained in a single container (e.g., a Python process) or as distributed jobs that can request multiple containers (e.g., a Spark ETL pipeline operating on DataFrames). In both cases, the AM is exclusively responsible for the execution and management of the life cycle. It is run in a container itself, provided, and spawned by the RM. (Vavilapalli et al., 2013)

The AM sends periodic heartbeat messages to confirm that it is still alive and to request containers with constraints mentioned in Section 4.1.2.5. The RM, in turn, returns container lease tokens to

the AM. Depending on the received resources, the AM can alter its physical execution plan to accommodate the acquired containers. Based on the physical execution plan, the AM uses the lease tokens to spawn containers directly from the NMs. Therefore, it can optimize the distribution of tasks depending on the locality of residing data and the available containers. (Vavilapalli et al., 2013)

In contrast to MapReduce, the job tracking is now part of the application or AM. This means that the RM does neither provide nor hinders monitoring and status updates of the application. This is either supplied by the AM or within the code of the application itself. As the RM does not interpret the container status provided by the NM, the AM is solely responsible for supervising the running and exit statuses of the containers for the computation. Application semantics and fault tolerance are tightly coupled, which renders the AM responsible and accountable to guarantee tolerance against failing containers. When the AM determines the application as finished, the RM receives the permission to release the granted containers and frees resources. (Vavilapalli et al., 2013)

NodeManager: The NM has the most work to do for running a job, second to the application itself. It manages the authentication of lease tokens, dependencies among containers, tracks their executions, and provides a set of auxiliary services for running a job. (Vavilapalli et al., 2013)

A so-called container launch context describes the commands to launch a container and defines environment variables to be set. Additionally, it lists remotely stored dependencies and contains payloads for NM services. With this information, the NM copies data files, scripts, configurations, and authentication credentials to a temporary, job-specific directory on its local file system. It then starts the container and initializes its application-specific monitoring system. The NM can also kill running containers if requested by the AM or RM. When the application is marked as finished, the NM will kill used containers, clean up the working directory on the local file system,

and terminate any still running processes, started by the application. (Vavilapalli et al., 2013)

Next to managing jobs, the NM also takes care of health statuses, concerning its managed resources, which are provided by the node machine. This entails frequently running an admin configured script, which points out hardware failures or software misconfiguration. If such an issue is discovered, it updates the RM about its unhealthy status through the heartbeat messages. The RM adapts its catalog of available resources and marks the containers provided by this NM as unhealthy, resulting in not giving out resources for the affected NM anymore. (Vavilapalli et al., 2013)

Spark on YARN: Spark shares a lot of architectural concepts with YARN, which makes it easier to align Spark's application workflow with YARN's provided functionalities.

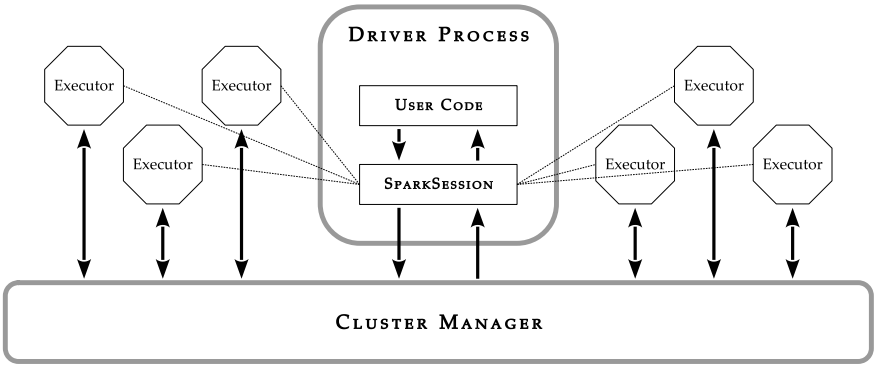


Figure 4.6.: Anatomy of Running a Distributed Spark Application - Based on Figures Provided by Chambers and Zaharia (2018)

Figure 4.6 describes the basic architecture of a distributed Spark job. Similar to the principle of running an AM which requests and manages worker containers on YARN, Spark runs a driver process that is responsible for interpreting, scheduling, and distributing computation work among its executors (workers, synonymously used to YARN's containers). The driver process is running the main code,

which can be the main() class of a Java application or a script of Python / R code. The process maintains and manages all relevant information about the running job. The executors do the actual distributed computation, for the most part, directly in a JVM, which runs compiled Java byte code from Spark's libraries. (Chambers & Zaharia, 2018)

It is important to stress here that, even if the client's application is written in Python, the workers execute Scala code (in the case of high-level APIs). Aside from the actual processing, executors have also to report their current and final computational states back to the driver process. (Chambers & Zaharia, 2018)

Spark on YARN provides two different deployment scenarios. Cluster mode initializes the job, acquires a container, and starts the driver process within this container on the cluster. This enables fail-safe execution of the application, if the driver process fails or the initializing client process disconnects. Client mode starts the driver application directly on the machine where the job was initialized, which results in a single point of failure if the local process fails. Though, it still needs to start an AM process in a container so that the same amount of containers are necessary as for cluster mode. (Apache Software Foundation, 2018d; Chambers & Zaharia, 2018)

4.1.2.6. Language Binding APIs

Apache Spark is mainly implemented in Scala, as described in Section 4.1.2.1, and feels therefore at home in the JVM-based Hadoop environment. It can integrate with HDFS and other JVM-based services of Hadoop natively. Though, Spark is able to work with a multitude of different sources, formats, and types.

As mentioned in Section 4.1.2.5, a driver process of Spark can be run in another language than Scala. Aside from Scala, Spark is usable with Java, Python (PySpark), and R. This does not mean, that Spark has been translated to the before-mentioned languages,

but rather provides language-specific API bindings for most of its functionality. This offers the possibility to combine Spark's high performance, distributed computation engine with the rich data-centric ecosystem of Python, R, and Java. (Karau & Warren, 2017)

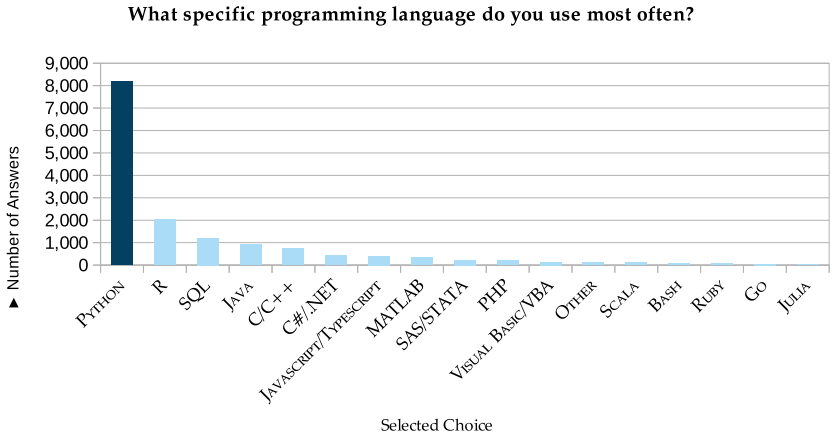


Figure 4.7.: Most Used Programming Languages for Data Science and Machine Learning - Based on Data Provided by Crawford et al. (2018)

Crawford et al. (2018) state on kaggle.com that their 2018 survey among data scientists and data engineers represents "[t]he most comprehensive dataset available on the state of ML [machine learning] and data science". As this audience corresponds well to the primary user base of Spark, this dataset gives us a good guideline on preferred programming languages for data-intensive tasks. Kaggle.com asked people through their channels and received 23,859 valid responses. The most interesting question for this thesis is "What specific programming language do you use most often?," which answers are visualized in Figure 4.7. It clearly shows that Python is undoubtedly the leading programming language used by data-centered practitioners. This thesis will therefore focus on the Python language binding of Spark and leaves out details about the Java, SQL, and R interfaces.

The SparkSession acts as the entry point to Spark. The Python API binds classes, objects, and methods in the Python environment to a

SparkSession running in a JVM. Spark uses PY4J for this, which acts as a bridge between the Python interpreter and Java, as PY4J can not directly communicate with the Scala SparkContext or interpret Scala collections. It is required to use a Java-friendly wrapper around the Scala SparkContext, which is called JavaSparkContext. Using high-level APIs of Spark allows converting Python commands to binary code, which runs in Spark executors. Transformations on PySpark DataFrames are executed in JVM byte code, whereas low-level data transformations on RDDs, using Python libraries, are executed within separate processes with their native compiler. (Nandi, 2015)

Figure 4.8 shows the extra effort of piping data objects between Scala and Python processes for low-level computations on RDDs. This includes translating Scala objects to Python objects, pipe them to another process, do the transformation, convert them back to JVM objects, and eventually pipe them back to the Spark context of origin. The performance will therefore suffer heavily when an application uses Python code for RDD operations. (Drabas & Lee, 2017)

Using a higher-level API, in contrast, leads only to minor performance losses. A common use case is to handle big data chunks with PySpark higher-level APIs and collect the small result as a Pandas data frame to process it further or visualize it. This handy Spark-DataFrame-to-Pandas conversion function is a built-in functionality of PySpark. The open-source community of Apache Spark is continuously working on additional integrations. Committers of the project are actively working on better interoperability of Python/Pandas with Spark. For example, support for vectorized user-defined functions (UDFs), written in Python, (SPARK-21190) is tracked at https://issues.apache.org/jira/browse/SPARK-21190. The umbrella ticket at https://issues.apache.org/jira/browse/SPARK-22216 gives a good overview of what is in progress at the moment. (Chambers & Zaharia, 2018; Nandi, 2015)

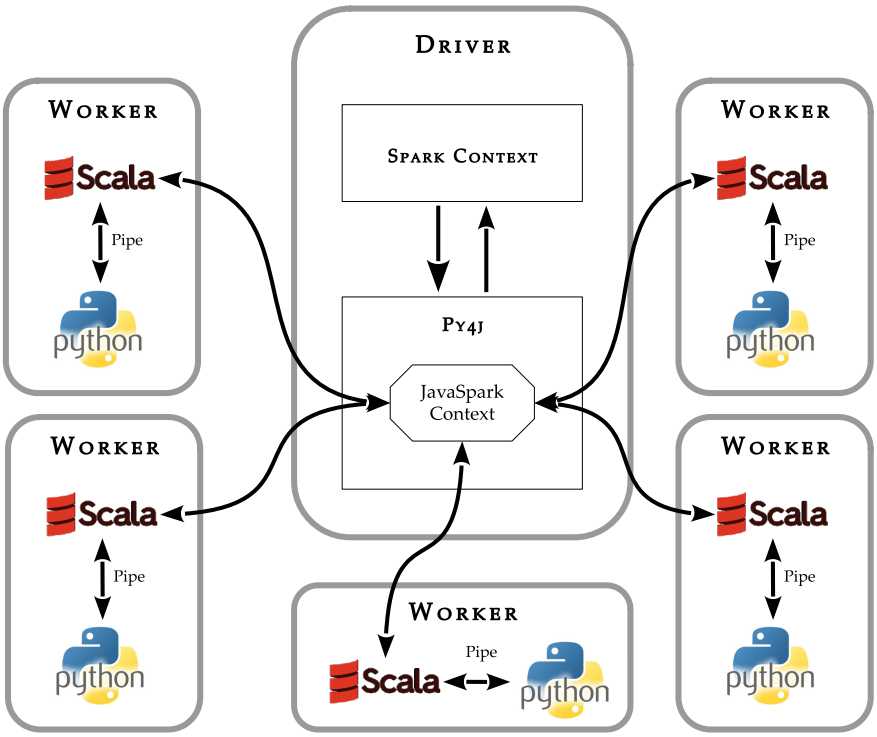


Figure 4.8.: Low-Level Processing with PySpark on RDDs - Based on Figures Provided by Drabas and Lee (2017)

4.1.2.7. Spark SQL

Using RDDs as its backbone allows Spark to enable a multitude of higher-level structures and APIs. One of the most important ones out of those is Spark SQL. This API, with its underlying DataFrames, is based on RDDs with fixed schemata. Spark SQL is often used for ETL/ELT tasks and other data wrangling jobs, and will, therefore, be described in more detail within this section. Datasets, which are also strongly typed, structured collections will not be part of this thesis, as they are not available in Python because of the language's lack of strong typing. (Chambers & Zaharia, 2018; Karau & Warren, 2017)

Introduced in Spark 1.3, DataFrames allow complex, SQL-esque queries on data gathered from different sources, such as CSVs, JSON

files, or databases via JDBC connections. Their inheritance from RDDs allows them to provide the same lazy evaluation, fault-tolerance, partitioning, and persistence as their low-level base. Streaming and batch processing are both compatible with Spark SQL. The fundamental concept of Spark distinguishes transformations from actions, which are both inherited by DataFrames. This results in constructing a DAG (directed acyclic graph) for the sequence of all transformations when an action is called. Only then, the graph will be analyzed, optimized, and executed as a single job, broken down into tasks and stages. (Armbrust et al., 2015; Chambers & Zaharia, 2018; Karau & Warren, 2017; Nandi, 2015)

A DataFrame can be viewed as a table-like data structure with an enforced schema, which has well-defined columns and rows. This means that a row in a DataFrame has values for every column, although they can be null. Furthermore, the data type of all values within a specific column must remain the same. Luckily, type inference takes mostly care of this constraint. Rows in DataFrames are sequences with special, Spark exclusive, data types, distinct to Scala's built-in data types, to optimize their in-memory footprint and efficiency of distributed processing. Therefore, it makes no difference if the driver application is written in Scala or Python as the execution is done purely in Spark, without any data type conversion needed — except for collecting a DataFrame to the driver process or UDFs based on Python code. (Armbrust et al., 2015; Chambers & Zaharia, 2018; Karau & Warren, 2017; Nandi, 2015)

Columns allow for simple data types like strings, floats, or timestamps, but also complex data types like arrays, structs, and hashmaps. This differs from the mostly flat structure of standard SQL engines like MySQL or SQL Server by Microsoft. A row is a distinct Spark SQL object which represents a collection of values of a specific Spark SQL data type, defined in the DataFrame's schema. They can be created from RDDs, files, external data sources, or by hand. A local — in the sense of non-distributed — Python Pandas data frame can easily be converted to a distributed Spark SQL DataFrame in one line

of code using Spark's Python binding API. (Armbrust et al., 2015; Chambers & Zaharia, 2018; Karau & Warren, 2017; Nandi, 2015)

Executing a data pipeline, based on DataFrames, runs through four steps as described by Chambers and Zaharia (2018):

1. A DataFrame or SQL command gets defined, written, and executed in a user-chosen language on the driver process.
2. Spark validates the code and constructs a logical plan, if it is executable.
3. The logical plan gets analyzed, optimized, and converted to a physical execution plan.
4. The physical plan, which consists of RDD transformations and actions, gets executed on the cluster.

The optimization of the actual processing is done by the Catalyst Optimizer. Figures 4.9 and 4.10 sketch the typical steps carried out by the code analyzer and the optimizer.

Before the execution plan reaches Catalyst, the logical plan constructed for a Spark SQL pipeline needs to be resolved. Even if the code and its syntax are valid, the analyzer still needs to check if all used tables and DataFrames are available. For this, it uses Spark's catalog, which holds the information for all sources and intermediate DataFrames and their corresponding columns, including data types. If the analyzer does not find any violations, the plan gets marked resolved and is passed over to the Catalyst Optimizer. A collection of rules are applied to the resolved plan, which results in predicates, able to be pushed-down the execution sequence. A basic example would be for a table to be loaded with ten columns, a group-by function is called on two attributes, and the distinct count on attribute three should be calculated. Catalyst would, in this case, adapt the loading command, that it only loads the three necessary columns. (Chambers & Zaharia, 2018; Drabas & Lee, 2017; D. Lee & Damji, 2016)

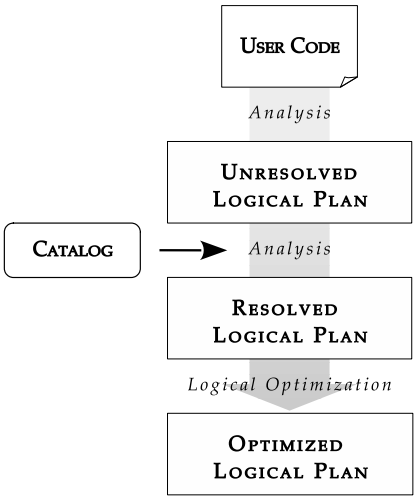


Figure 4.9.: The Catalyst Optimizer Logical Plan - Based on Figures Provided by Chambers and Zaharia (2018)

As the next step, with the optimized logical plan as input, the physical planning process begins. Therefore, Catalyst constructs different physical plans, evaluates them through a cost-based comparison, and chooses the most efficient. This can make a significant difference, depending on the size and how the partitions are distributed. The output of this process is a plan of RDD transformations, ready to be translated to Java byte code and executed efficiently as opposed to defining the RDD transformations manually. This holds especially true for applications written in Python or R. (Chambers & Zaharia, 2018; Drabas & Lee, 2017; D. Lee & Damji, 2016)

Project Tungsten should also be mentioned in this section as it provides similar performance increases to Spark applications as the Catalyst Optimizer, although not exclusively for Spark SQL. This is achieved by binary processing, explicit memory managing, cache-aware computation, and byte code generation at compile time rather than at execution time. (Chambers & Zaharia, 2018; Drabas & Lee, 2017; D. Lee & Damji, 2016)

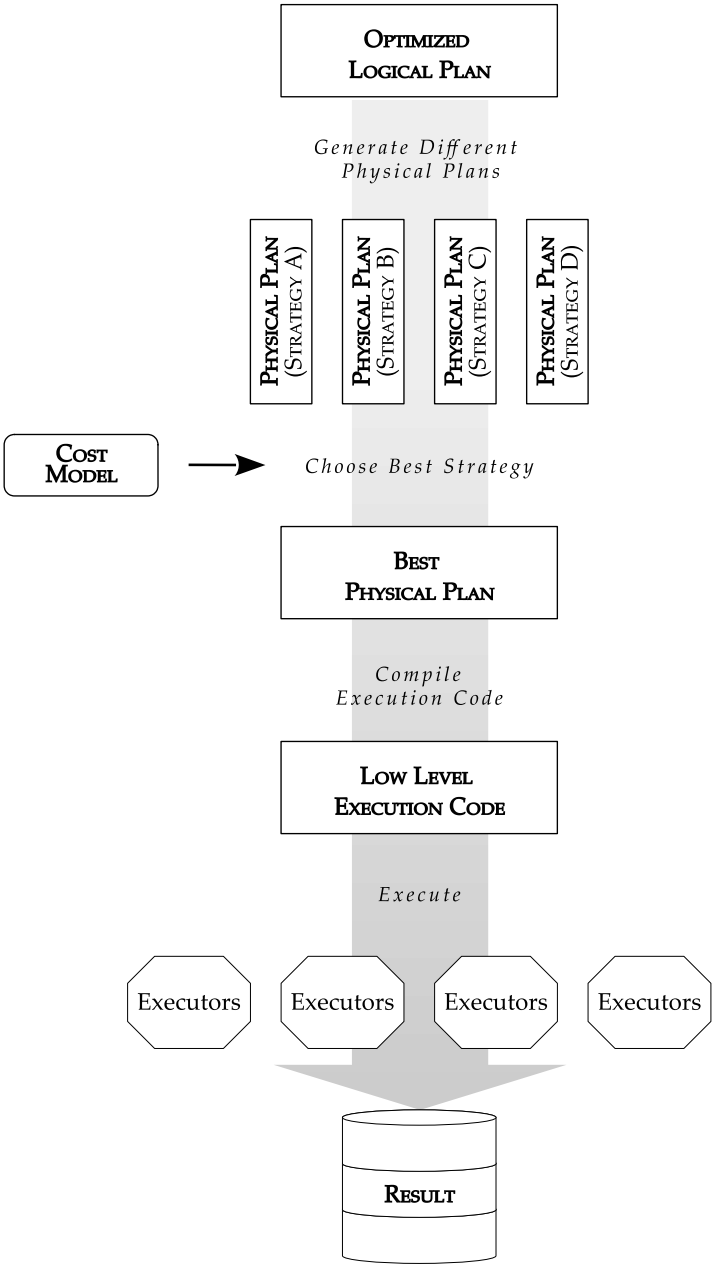


Figure 4.10.: The Catalyst Optimizer Physical Plan - Based on Figures Provided by Chambers and Zaharia (2018)

4.1.3. Expert Systems

Spooq tries to decrease the complexity of compiling ETL pipelines with the help of an expert system. This section introduces the reader to the basics of expert systems. Section 4.1.3.1 continues by describing knowledge bases, which are used by expert systems to utilize formalized information, and gives some examples on how to design them. Inference engines — the logical part of expert systems — are explained in Section 4.1.3.2. *Experta* — the rule-based expert system that was used for the artifact of this thesis — is covered in Section 4.1.3.3.

Feigenbaum (1981) defines an expert system as

> "... an intelligent computer program that uses knowledge and inference procedures to solve problems that are difficult enough to require significant human expertise for their solution. The knowledge necessary to perform at such a level, plus the inference procedures used, can be thought of as a model of the expertise of the best practitioners in that field."

ES (expert systems), also sometimes referred to as KBS (knowledge-based systems), production rule systems, or simply production systems, are software-based constructs that try to solve problems posed by humans. They are a branch of Artificial Intelligence and became popular in the 1980s, when first commercial implementations came to the market. Expert systems are characterized by the functionality of solving complex problems, relying mainly on formalized, specific expertise, in a limited field of knowledge. General and shallow problem-solving systems, in contrast, try to be applicable in a wide area of problem space. (J. Giarratano & Riley, 2005; Sasikumar et al., 2007)

An expert system acts similar to a domain expert in providing expertise to users, based on available information, specific to their situation. The general mode of operation of an ES is illustrated in Figure 4.11.

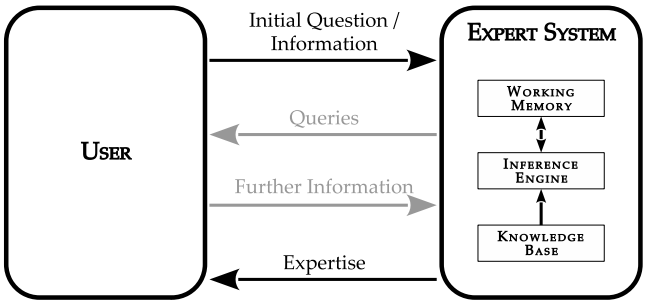


Figure 4.11.: General Mode of Operation of an Expert System - Based on Figures Provided by Sasikumar et al. (2007)

A human actor requires expertise on a specific problem or situation and poses his/her question to an expert system while providing the ES with information about the problem. This request has to be formulated in the form of *facts* for the ES to be able to understand it. The inference engine gathers domain-specific rules and validates them based on the provided facts. Outcomes of rules can be modified, new, or deleted facts, which makes reevaluation necessary. The dynamic list of facts and rules are kept within the working memory of the expert system. In the case of an interactive expert system, the user can answer questions or provide additional information about the problem (illustrated in gray color) as required by the ES to evaluate further rules. When no more rules are satisfied, or an explicit stop is triggered, the ES returns its inferred conclusions, in the form of problem-specific expertise, back to its user. (J. Giarratano & Riley, 2005; Sasikumar et al., 2007)

Compared to human consulting, several advantages are generally associated with expert system-based problem solving. The availability of such a problem-solving instrument is not limited in any way, provided that appropriate hardware is available. Multiple users can work on it simultaneously, and results are returned almost immediately. Having expert knowledge in formalized formats makes it independent of human actors who can get sick, retire, or change their jobs. Costs are reduced as an expert opinion does not take any time nor resources from a domain expert. The outcome is determinis-

tic, whereas human experts can come to different conclusions based on their current state — for example, when they are tired, sick, or stressed. Emotions will not interfere with the conclusions of an ES. Rule-based ES are easier to interpret as they base their deductions on formalized rules, fed by facts. Expert systems can show which rules were triggered due to what facts to explain a resulting conclusion. AI systems based on machine learning, like artificial neural networks, are, on the other hand, often more difficult to interpret. Human experts are nonetheless crucial for expert systems as they provide the knowledge base from which the inference engine can deduct information. (J. Giarratano & Riley, 2005)

The next section will go into more detail about the knowledge base, its content, and how it is created and maintained.

4.1.3.1. Knowledge Base

A knowledge engineer gathers knowledge from a domain expert and persists it into formal structures. The dialog between domain specialists and engineers is of utmost importance to avoid translation mistakes in this error-prone conversion. Domain experts often do not know — or do not have the resources — to expatiate their knowledge in a machine-readable form. Knowledge engineers, on the other hand, often do not have enough understanding of the problem and solution space at hand. Once the expertise of domain professionals is transferred to the engineers, a knowledge base can be created, which is fed by this information. Buchanan et al. (2006) categorize this information into factual and heuristic knowledge. Factual knowledge refers to common knowledge about a domain that can be found in papers, books, and other commonly agreed-on media which is based on research. Expertise that is based on the experience of practitioners in the problem domain is classified as heuristic knowledge, which is often of individualistic and subjective kind. Heuristic knowledge contains best practices, individual reasoning, and implicit knowledge of the domain experts. There are several ways to manifest this extracted intelligence. (Sasikumar et al., 2007)

Semantic nets store their information as directed graphs with labeled edges. Each node represents an atomic fact, like *"All squares are rectangles"* or *"A rectangle has four right angles"*. Depending on its evaluation, other nodes can get activated. This linked logic is described by different kinds of directed edges, like *"IS-A"* or *"HAS-COLOR"*. Semantic nets add a relationship aspect to a collection of otherwise disconnected data points to show and use its associative information. (J. Giarratano & Riley, 2005)

OAVs (Object-Attribute-Value Triplets) limit the open approach of semantic nets to mitigate the sometimes confusing ambiguity of links. Only nodes of type object, attribute, and value are allowed for OAV-based semantic nets. Restricting the edge types between objects and attributes to *"HAS-A"* and links between attributes and values to *"IS-A"* results in a well-defined structure, which is also easy to persist in a relational table. (J. Giarratano & Riley, 2005).

Other note-worthy types used for knowledge representation are *frames* and *formal logic*. *Frames* are a special form of schemata, which add support for generic abstractions of objects. *Scripts* enrich the concept of frames by a temporal dimension, in such that scripts behave like chronological series of frames. *Formal logic* relies on premises and conclusions. Its facts and information are connected by causality. If one rule's conclusion satisfies another premise, another conclusion can be drawn. In its simplest form, syllogism infers a conclusion from two premises. (J. Giarratano & Riley, 2005)

An example about the mortality of Socrates, taken from J. Giarratano and Riley (2005), is given below in the form of a syllogism:

```
PREMISE:    All men are mortal
PREMISE:    Socrates is a man
CONCLUSION: Socrates is mortal
```

The inference functionality of *spooq_rules* is based on production rules. This type of knowledge representation is one of the most common forms to structure information in expert systems and is described in more detail in the following paragraphs.

In the form of a production rule-based ES, simple IFTTT (IF This Than That) rules are the formalized output of a knowledge engineering process. Several languages were created and used to express those logical rules. The generic structure consists of *antecedents* and *consequents*. *Antecedents*, also called patterns or LHS (left-hand side) of a rule, describe conditions that are to be evaluated against a list of facts. They contain requirements that have to be met to trigger the *consequent* of a rule. A rule in its most basic form consists of one condition and action. (Sasikumar et al., 2007)

Here is an exemplary duck test, expressed as a simple production rule:

```
IF (Antecedents):
    X walks like a duck
THEN (Consequent):
    X is a duck
```

Depending on the implementation of an expert system, rules can also be described in more complex manners. An example of such a rule is shown below by an abstraction of the decision tree for the iris flower classification by McRitchie (2018):

```
SALIENCE (Priority):
    3
IF (Antecedents):
    Petal.Length > 5.1cm
    OR IF
        Petal.Length between 2.5cm and 5.1cm
        AND Petal.Width > 1.8cm
THEN (Consequent):
    species = Virginica
CF (Confidence Factor):
    0.8
```

In addition to quantitative conditions, the iris flower classification rule also uses a *salience* (priority) setting for the case of conflict resolution and a *CF* (confidence factor) to allow for non-absolute conclusions. (Sasikumar et al., 2007)

Once a sufficiently equipped knowledge base is created, domain experts and professionals of the problem's solution space can evaluate its performance. J. Giarratano and Riley (2005) list improved accuracy and increased trust of operators towards the expert system as benefits of frequent in-house evaluations by knowledge engineers and experts. Ultimately, for the ES to be able to derive conclusions, it needs to apply an inference algorithm upon the data stored in the knowledge base. The next section gives an overview of different inference techniques and describes the performance-improving *Rete Pattern-Matching Algorithm* in more detail.

4.1.3.2. Inference Engine

An inference engine is the component of an ES which generates new information from its input and available data. It reasons about given facts with the details of the knowledge base and returns its conclusions. There are several algorithms and logical methods to infer new knowledge from existing one. For an expert system, conclusions are often made through *induction*, *deduction*, and *abduction*. J. Giarratano and Riley (2005) also mention alternative inference processes which are supported by intuition, heuristics, trial and error, common knowledge, autoepistemic knowledge, nonmonotonic knowledge, and analogies. (Douven, 2017; Sasikumar et al., 2007)

Concluding a fact about a particular entity based on general knowledge is called *deduction*. An example would be: *"This pack of gummy bears only contains red items, as stated on the wrapping. Therefore, a randomly drawn gummy bear will be red."* The probability of deduced facts relies mainly on the general rule's certainty of truth. If a premise is true, the conclusion is necessarily true as well. *Induction* describes the way of inferring a generalized rule from statistical data. Deriving generalizations, based on a limited sample size of observations, however, always involves an element of probability. An induced conclusion about drawing gummy bears would be: *"97% of all sold gummy bears packages in the last year exclusively contained red items. A*

randomly picked gummy bear from a blindly bought package will be of color red." The premise's probability of truthiness has a ripple effect on the conclusion drawn from it, which makes the outcome non-necessarily true. Rather than inferring facts from statistical data like induction, *abduction* is about drawing a conclusion from an observation which is supported by selective and probable premises. *Abduction* is also referred to as *inference to the best explanation* which emphasizes that this form of reasoning tries to find an explanation of the observation which is not necessarily true but likely. This method is sometimes the only way of deriving facts from incomplete or unclear data. The resumed example would read: *"I left my pack of gummy bears on the kitchen table over night. The next day the pack was gone and the dog had nausea. Therefore the dog ate the pack of gummy bears."* Even though the dog is able to reach the kitchen table and gummy bears can cause nausea for dogs, it could also be that a roommate ate the pack when he or she got up at night and the dog's sickness is purely coincidental. (Douven, 2017)

The use case of inference for *Spooq* is to derive designs and parameters for data pipelines. The problem and solution space is rather small and very well understood, which leads to strong premises with high certainty. Drawn conclusions are specific to an instance of an ETL/ELT process, and therefore, there is no need to derive generalized knowledge from the facts. *spooq_rules* uses deduction supported by a production rules-based expert system.

Production rules are abstractions of logical correlations between facts with certain attributes and triggered actions, as described in Section 4.1.3.1. They can be seen as a distinct type of generalized knowledge, suitable for expert systems. Facts are instantiated information about individual entities. In the case of a rule-based ES, facts are instantiated conditions of a rule, given that a rule exists which coincides with the fact's information. If an entity, described via a fact, satisfies the antecedents of a rule, the consequents get activated. (Sasikumar et al., 2007)

To resume the example from before, a production rule about Socrates' mortality would be formalized as follows:

```
RULE:           IF X is a man  THEN X is mortal
INITIAL FACT:   Socrates is a man
INFERRED FACT:  Socrates is mortal (conclusion)
```

When a consequent of a rule creates new facts, additional rules can get activated, and previous ones can get deactivated. Stating a single fact can lead to a sequence of inferred facts, which ultimately yields facts that answer the user's question. This cyclic process, from initial facts, over inferred facts, to a final conclusion, is usually called a chain. There are generally two ways to traverse such a series of inferences, *forward-chaining* and *backward-chaining*. (J. Giarratano & Riley, 2005; Sasikumar et al., 2007)

Rule-based expert systems are most often used for deductive inference, following a forward-chaining paradigm. Instead of explaining or validating a hypothesis, they try to generate conclusions based on low-level facts (in terms of complexity). Bottom-up is an alternative name for forward-chaining, because the initial facts, where the chain starts to infer, represent the atomic, bottom level of information. Further down the chain, facts can become higher-level and more specific. This type of reasoning is generally driven by its a priori knowledge towards a goal. (J. Giarratano & Riley, 2005)

A forward-chaining, causal inference chain, expanding the example of Socrates, is shown here:

```
RULE 1:  IF X is a man     THEN X is human
RULE 2:  IF X is a human   THEN X is a mammal
RULE 3:  IF X is a mammal  THEN X is mortal

INITIAL FACT:  Socrates is a man

INFERRED FACT:  Socrates is a human
(outcome of RULE 1)
INFERRED FACT:  Socrates is a mammal
(outcome of RULE 2)
INFERRED FACT:  Socrates is mortal
(outcome of RULE 3 / conclusion)
```

Backward-chaining or goal-driven search inverses the path and tries to come up with facts that validate or confirm a hypothesis or goal. Sub-goals are temporarily created in the process to potentially complete a logical link from the provided facts to the end goal. This top-down procedure is best suited for connecting the present with the past. For example, diagnosing a patient depending on his or her anamnesis or finding explanations for a decision. By following the link from consequents to antecedent, a given conclusion results in evidence to support it. Backward-chaining is driven by a given goal towards low-level facts. (J. Giarratano & Riley, 2005)

In the case of the example above, the starting point would be the question if Socrates is mortal and the resulting facts, proving the mortality, would be presented to the inquirer via sub-goals and inferred facts:

```
RULE 1:  IF X is a man     THEN X is human
RULE 2:  IF X is a human   THEN X is a mammal
RULE 3:  IF X is a mammal  THEN X is mortal

GOAL (HYPOTHESIS):  Is Socrates mortal?
(corresponds to consequent of RULE3)

SUBGOAL1:  Is Socrates a mammal?
(follows antecedent of RULE3)
SUBGOAL2:  Is Socrates a human?
(follows antecedent of RULE2)
SUBGOAL3:  Is Socrates a man?
(follows antecedent of RULE1)

INITIAL FACT:  Socrates is a man
(satisfies SUBGOAL3)
INFERRED FACT:  Socrates is human
(satisfies SUBGOAL2)
INFERRED FACT:  Socrates is a mammal
(satisfies SUBGOAL1)
INFERRED FACT:  Socrates is mortal
(satisfies GOAL)

CONCLUSION:  Yes!
```

Logical chains of reasoning imply that there are several intermediate phases of an inference process. When an inference engine is started, initial facts are loaded into the working memory. Rules from the knowledge base get evaluated to see if any facts within the working memory satisfy their antecedents. This leads to a collection of applicable rules, called the conflict set. The order in which rules are triggered makes a difference, as each rule can alter the working memory and therefore the set of valid rules. The resolution strategy influences the sequence in which the inference engine performs the actions. There are several aspects upon which an algorithm can choose the course of action. (Sasikumar et al., 2007)

Specificity evaluates the complexity of a rule's antecedents. The more conditions a LHS contains, the more specific a rule is classified and the higher its priority gets. *Recency* favors rules which are satisfied by more recent facts. *Refraction* simply takes care of how many facts can trigger a single rule to avoid undesired loops. An ordered conflict set, also called an agenda, defines in which course of action rules get executed. (J. C. Giarratano, 2015; Sasikumar et al., 2007)

Most expert systems provide a resolution approach which combines several factors and calculations. When an agenda is compiled, each rule's action is performed until the working memory is altered by adding, modifying, or deleting a fact. A changed working memory requires to re-evaluate the rule's conditions again, and the so-called *recognize-act cycle* is started again. The inference process stops when no applicable rules are left, or an explicit halt is called. (Sasikumar et al., 2007)

Evaluating facts against the LHS of rules is a costly computation. Considering that knowledge bases can grow to considerable sizes, and each fact has to be compared to each condition from each rule, most of the computing time of an ES is spent on rule matching. Moreover, the pattern matching process has to start again after each alteration of the working memory, which increases the runtime considerably. A lot of different solutions have been developed to mitigate this performance bottleneck. The *Rete Pattern-Matching Algorithm* is one of

the best-known procedures, used by most rule-based expert systems, including *spooq_rules'* inference library. (Sasikumar et al., 2007)

The *Rete Pattern-Matching Algorithm* was developed by Forgy in 1979. The most naive modus operandi of an inference engine is to iterate over all rules and validate facts that correspond to their LHS to see if they are applicable, as shown in Figure 4.12. Every time the working memory has changed, all rules have to be re-iterated, although only a few rules are affected. (Sasikumar et al., 2007)

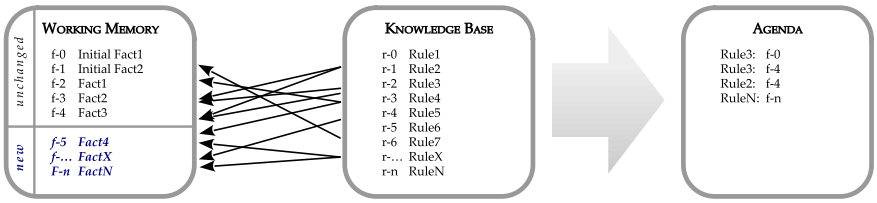


Figure 4.12.: Redundant Pattern Matching When Rules Search for Facts - Based on Figures Provided By J. Giarratano and Riley (2005)

The rete algorithm makes heavy use of the *temporal redundancy* concept, which takes into account that a rule's consequent usually changes only a small portion of the working memory. The general idea is to reverse the direction from rules-towards-facts to facts-towards-rules. Figure 4.13 illustrates the approach taken by the temporal-redundancy-aware rete algorithm with respect to the number of patterns to match. (Forgy, 1979; J. Giarratano & Riley, 2005)

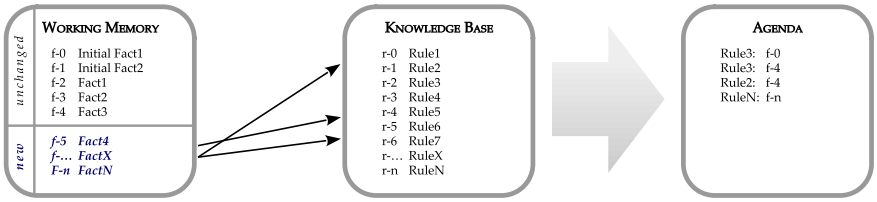


Figure 4.13.: Efficient Pattern Matching When Altered Facts Search for Rules - Based on Figures Provided By J. Giarratano and Riley (2005)

Another property of production systems is the *structural similarity* of the rules' LHS. An ES knowledge base often consists of rules which share a substantial amount of (sub-)patterns among them. The antecedents of production rules usually consist of conditions or field constraints, which are combined by logical conditional elements like `AND` or `NOT`. Keeping an overview of (sub-)patterns and their corresponding rules in memory, allows a rete-based inference engine to exclude rules without evaluating them. For example, one rule reads: *"IF A THEN C"* and a second rule defines: *"IF A AND B THEN D"*. Determining the first rule as inapplicable allows to conclude that the second can not be applicable either, as it contains the same, already falsified antecedent within an `AND` condition. (Forgy, 1979)

The next section will go into detail about the syntax and usage of *Experta*. This python library is used by *spooq_rules* to implement its reasoning for the automatic generation of *Spooq* pipelines.

4.1.3.3. Experta

spooq_rules uses a production rule-based library, named *Experta*, for its inference of parameters to construct data pipelines. This Python library is a fork of *PyKnow*, which is heavily inspired by *CLIPS* (C Language Integrated Production System).

CLIPS was developed at NASA/Johnson Space Center with the primary goal of providing rule-based production systems to increase efficiency, portability, and integratability. The programming language introduced support for object-orientation and procedural capabilities in Version 5.0. The last stable release was in 2015 with Version 6.30. *Experta* tries to be as compatible as possible to make it easier for knowledge engineers coming from *CLIPS* to migrate their knowledge to *Experta*. (J. C. Giarratano, 2015; Pérez, 2019)

Experta supports programming through rules, objects, and functions, like *CLIPS*. Due to the fact that *Experta* is a pure Python library, in contrast to *CLIPS* as a programming language, a few limitations occur

— mainly, comparably lower performance and some restrictions in defining antecedents of rules. (Pérez, 2019)

Both expert systems (*Experta* and *CLIPS*) consist of three components comparable to Figure 4.11. A *fact-list* acts as the working memory and contains initial and inferred information in the form of facts. The *knowledge base* consists of data in the form of rules. An *inference engine* takes care of validating facts upon given rules. (J. C. Giarratano, 2015; Pérez, 2019)

Facts: Facts are envelopes in which data is transported. Those units can contain arbitrary information. The object-oriented architecture of *Experta* allows for facts in the form of class instances. A simple fact object is a specialized form of a Python dictionary. It can be easily constructed by passing key/value pairs to the initialization method, like `Fact(a=1, b=2)`. Storing and accessing values via an index, without a key to reference them, is also supported. This can be combined with named values if they are defined after the indexed ones, like `Fact('x', 'y', 'z', a=1, b=2)`. Those facts are the basis for reasoning about a user's query by comparing them with a predefined ruleset. (Pérez, 2019)

Antecedents: A rule is composed of an antecedent and an action. The antecedent, also called pattern or LHS (left-hand side of a rule), is defined in *Experta* as a Python decorator to a function. When all conditions, formalized as antecedents in the LHS, are met, the function is called and can execute any code given. An example taken from *Experta's* documentation is given in Code Block 4.1 to demonstrate the basic syntax of a `Fact`.

Code Block 4.1.: Example of a Fact from Experta's User Documentation (Pérez, 2019)

```
1  @Rule(Fact('animal', family='felinae'))
2  def match_with_cats():
3      print("Meow!")
```

The rule's LHS expects a fact of type `animal` with the value *felinae* for the attribute `family`. When the LHS evaluates to true, *Meow!* is printed, as defined in the consequent. (Pérez, 2019)

The LHS of a rule can contain multiple conditional elements to combine several patterns in one rule. Next to basic elements like `AND`, `OR`, `NOT`, and `EXISTS`, also more complex ones can be used like `FORALL`, which compares multiple facts for a common value, and `TEST` which is used to evaluate an arbitrary condition formulated as a lambda function. With FC (field constraints), less precise analysis can be done on facts. `W` (wildcard field constraint) checks for any values on a specific attribute. `P` (predicate field constraint) applies a callable to a fact which evaluates to a boolean. All FCs can be chained together with logical `ANDFC` (&), `ORFC` (|), and `NOTFC` (~) to construct more complex rules. The `MATCH` command allows to bind a value of an attribute to a variable that is accessible within the context of the consequent's function. (Pérez, 2019)

Consequents: The consequent, which is executed when a rule is fired (when all conditions are met), can be rather simple, like printing a word to the console. The `MATCH` operation in the LHS transfers the fact into the context of the actions' function, which enables interaction with it. Expert systems are designed to have a dynamic state of facts, which allows the RHS of a rule to *declare*, *modify*, *duplicate*, and *retract* facts. Since *Experta* is realized in pure Python, a consequent's function can also interact with non-*Experta* Python code, like accessing a database or serving a REST API. (Pérez, 2019)

Knowledge Engine: *Experta* provides the class `KnowledgeEngine`, which consolidates the rule definitions with its inference engine, supported by a working memory. Code Block 4.2 illustrates the syntax to define a `KnowledgeEngine` of *Experta* through a complete, although simple, example from its user documentation. Please refer

to Code Block 4.3 for the application and usage of this example. (Pérez, 2019)

Code Block 4.2.: Example of a KnowledgeEngine Definition from Experta's User Documentation (Pérez, 2019)

```
from experta import *

class Greetings(KnowledgeEngine):
    @DefFacts()
    def _initial_action(self):
        yield Fact(action="greet")

    @Rule(Fact(action='greet'),
          NOT(Fact(name=W())))
    def ask_name(self):
        self.declare(Fact(name=input("What's your name? ")))

    @Rule(Fact(action='greet'),
          NOT(Fact(location=W())))
    def ask_location(self):
        self.declare(Fact(location=input("Where are you? ")))

    @Rule(Fact(action='greet'),
          Fact(name=MATCH.name),
          Fact(location=MATCH.location))
    def greet(self, name, location):
        print("Hi %s! How is the weather in %s?" % (name, location))

engine = Greetings()
engine.reset()  # Prepare the engine for the execution.
engine.run()  # Run it!
```

The execution of the example defined in Code Block 4.2 starts by setting the `action` to *greet* via yielding the initial fact. This is denoted by the `@DefFacts()` decorator and initiated by `engine.reset()`. `engine.run()` starts the inference process. As there are no priorities set, the top rule in the code will be processed first. The `action` attribute equals *greet* and the attribute `name` holds no value, which activates the action of the function. The user is asked about his name, which is sent to the working memory in the form of a fact by the `declare()` method. Please note that in *CLIPS*, the keyword *assert* is used to add new facts to the working memory, whereas in Python *assert* is a protected command. *Experta* uses therefore *declare* instead of *assert*. Every time the working memory changes, a new round of rule evaluation has to be performed. The first rule is excluded as a fact already triggered it and the new fact does not satisfy its conditions. The new fact, however, evaluates true to the second rule,

which sets the location. The third and last iteration fires the last rule, which prints a string to the console.

Code Block 4.3.: Example of a KnowledgeEngine Application from Experta's User Documentation (Pérez, 2019)

```
$ python greet.py
What's your name? Roberto
Where are you? Madrid
Hi Roberto! How is the weather in Madrid?
```

Experta's general sequence of action consists of a cycle with three phases. The first determines if the execution of the inference engine is finished or should stop. If no rules can be triggered anymore, the application will halt. The second phase collects and orders applicable rules in an agenda, which is then sequentially processed. As it takes note of added, changed, and retracted facts, previously disabled rules can become activated and vice versa. Priority settings (via the `salience` attribute) and different conflict resolution strategies can influence the order of the execution sequence in the agenda. The third action in *Experta's* processing-cycle is to call the rules' RHS, which are defined in their method bodies. (Pérez, 2019)

The simple example described in Code Block 4.3 requires three iterations, with three rule evaluations for each. This totals to nine assessments. However, for more practical applications, the number of rule evaluations can increase substantially, which led *Experta* to use the rete algorithm internally. The implementation is kept close to the original concepts of Forgy (1979), which is described in more detail in Section 4.1.3.2. Only adaptations in a few parts were made to uphold the parity to *CLIPS's* functionality. (Pérez, 2019)

4.2. Implementation

This section describes how *Spooq* is implemented. It starts with an overview of the architecture, illustrated by a data flow graph and UML class diagrams. Further on, pipeline, extractor, transformer, and

loader components are described in detail. The set-up of test cases and *Spooq's* documentation is explained later on. The last section goes into detail on *spooq_rules*, which provides the reasoning functionality for semi-automatic configuration of *Spooq* data pipelines.

The figures included in this section use following conventions to keep them consistent and avoid ambiguity. The stick figure represents a client, which can be a data scientist, data engineer, or a programmatic scheduler. Namespaces that are used to group different (sub-)classes are represented by stylized record files. Classes are marked by a capital C — capital A for abstract classes — within a circle with gray background color. Python modules are represented by a capital M within a circle with dark gray background color. Low cylinders symbolize database systems or file storages. Data in transit is depicted by a gray rectangle with the top right corner folded-in. Notes and comments are displayed in a white rectangle with the top right corner folder-in and are pointing to the commented object. Manual line breaks are marked with a carriage return symbol at the end of the line. Other symbols and arrows used in the UML class, data flow, and activity diagrams follow their respective standard notation provided by the *PlantUML* library, which was used to render the figures.

4.2.1. Architecture

Spooq is implemented in Python 2 as Python 3 is not supported in older Spark environments. Due to the requirements of Spark, at least version 8 of Java is required. For the development of *Spooq*, the free implementation of Java called *jdk8-openjdk* was used. All dependencies for development are managed by *pipenv*, which is a combination of *virtual environments* and Python's *pip* package manager, according to its creator, Reitz (2018). Setting up all necessary packages of *Spooq* is done by executing **`pipenv install -dev`**. To start developing, testing, and documenting, **`pipenv shell`** has to be called to enter the virtual environment for development.

The architectural design of *Spooq* revolves around the strict decoupling of individual components. Its domain specificity allows to define a fixed set of class-archetypes. Pipelines define the main flow of action, which is processed by its ETL members. Extractors, transformers, and loaders have well-defined interfaces that promote independence and interchangeability of their subclasses.

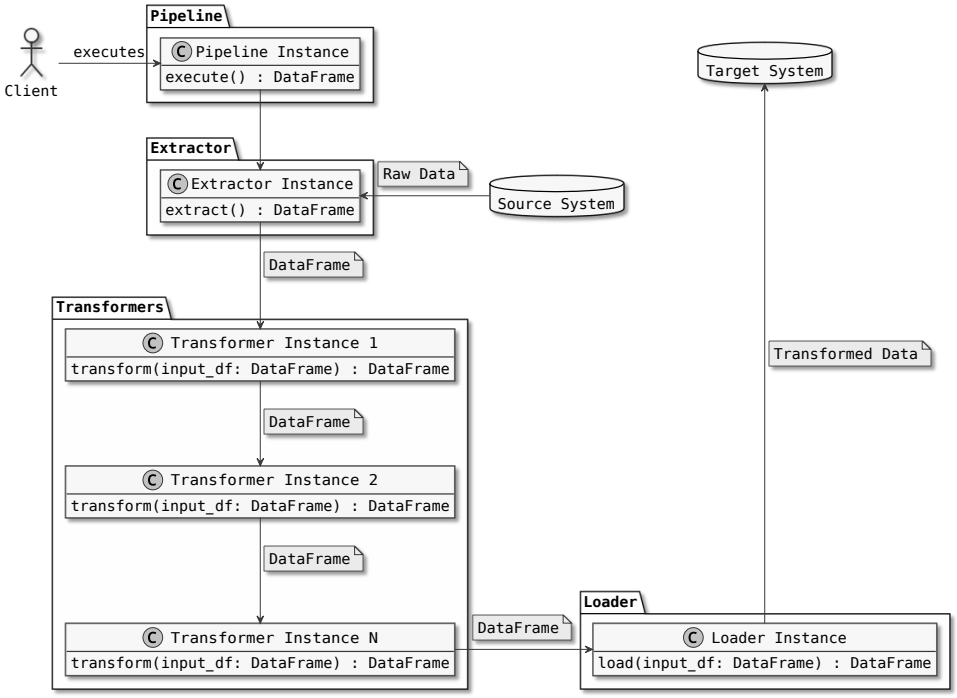


Figure 4.14.: Typical Data Flow of a Spooq Data Pipeline

Figure 4.14 visualizes the typical flow of data for a pipeline built with *Spooq*. A client start the processing by calling the `execute()` method. An extractor instance takes care of the extraction process and passes the external raw data as a PySpark DataFrame to the transforming subsystem. An ordered list of transformer instances is called sequentially, with each instance receiving the output DataFrame from its previous transformer. Their `transform()` method expects a DataFrame as its sole parameter and returns a single DataFrame, as per definition. In the last phase, the successfully transformed DataFrame

is passed to a loader instance. The `load()` function stores the dataset to a predefined target system, which concludes the data pipeline processing.

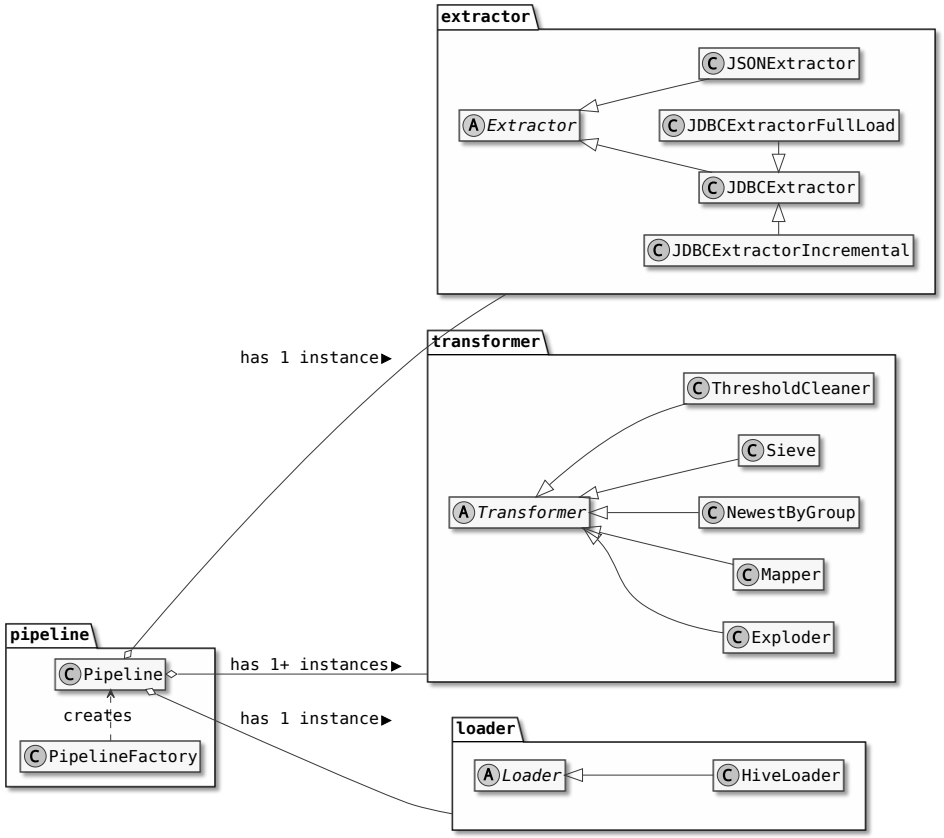


Figure 4.15.: Class Diagram: Spooq

Figure 4.15 gives an overview of the architecture of *Spooq*. A simplified UML class diagram portrays the four main subpackages of data pipelines built with *Spooq*. A pipeline instance contains exactly one extractor which is responsible for gathering, decoding, and converting data into a PySpark DataFrame. For the transformation, a list of transformer instances is kept within the pipeline object to process the extracted DataFrame sequentially. The number of transformers is not

limited apart from the necessity of at least one transformer instance. An exclusive loader instance determines the output location, format, and parameters and eventually performs the data persistence.

All definitions and configuration of the individual ETL components are passed to the objects at the initialization phase. This allows for parameter-less execution after a *Spooq* pipeline is constructed. Tying all parameterization to a single instance avoids inter-dependencies among the components and intra-dependencies within the pipeline. All components of type `Pipeline`, `Extractor`, `Transformer`, and `Loader` share the same `logger` instance, which is automatically set at the initialization phase.

A separate application can be used to enable nearly configuration-less operations of *Spooq*, which generates the necessary pipeline definitions automatically. This information is passed to *Spooq's* class `PipelineFactory` which constructs and executes a fully configured `Pipeline` instance. More details about this functionality follow in Section 4.2.8.

4.2.2. Pipeline

The `Pipeline` class is a top-level construct of *Spooq*, which contains all relevant components to perform a complete ETL or ELT process. As opposed to extractors, transformers, and loaders, a pipeline object is initialized with default attributes and later-on parameterized in runtime by adding instances of ETL components.

Figure 4.16 outlines the parameters, attributes, and public methods of *Spooq's* `Pipeline` class. The attributes `extractor`, `transformers`, and `loader` are empty after initialization to be set later via the methods `set_extractor()`, `add_transformers()`, and `set_loader()`, respectively. As the order of transformers can not be changed after they are added to the `Pipeline` object, a `clear_transformers()` method is available to reset the list of transformers.

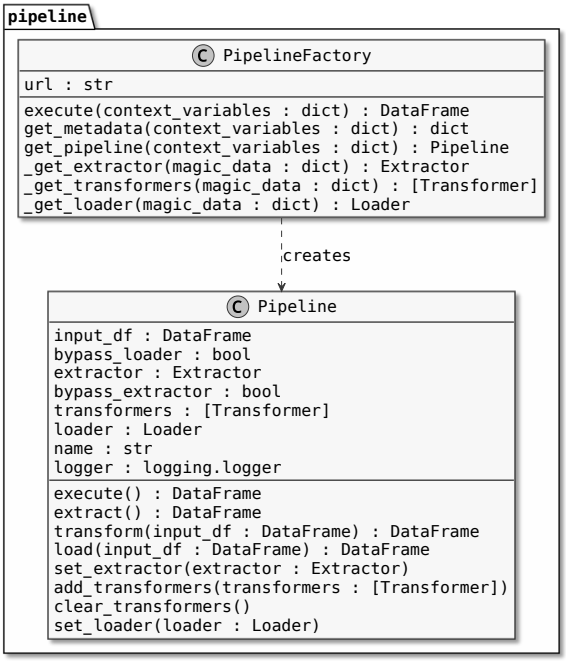


Figure 4.16.: Class Diagram: Spooq's Pipeline Subpackage

Providing a DataFrame (`input_df`) as a parameter to the Pipeline object sets the `bypass_extractor` to *True* and consequently skips the extraction process. The `extract()` method directly returns the supplied `input_df` DataFrame in this case.

A similar logic is applied to the loader instance. If the `bypass_loader` flag is set to *True*, the `load()` and `execute()` functions will return the processed output DataFrame instead of persisting it to a predefined sink.

The methods `extract()`, `transform()`, and `load()` can be either called explicitly or implicitly through the `execute()` function, which takes care of passing the DataFrames between the ETL operations.

Code Block 4.4 showcases a simple pipeline configuration with one extractor, transformer, and loader, each. After importing the component's subpackages as `E`, `T`, and `L`, a `Pipeline` object is initialized

without any parameters. A `JSONExtractor` instance, which points to a set of input sequence files, is set. In this case, the list of transformers only contains a single `Mapper` instance, which takes care of selecting, renaming, and casting columns from the extracted data. To finish the construction of the pipeline, a `HiveLoader` is set, which defines the output database and table name. Calling `execute()` on the `pipeline` object triggers the processing.

Code Block 4.4.: Example: Pipeline

```
>>> from spooq2.pipeline import Pipeline
>>> import spooq2.extractor as   E
>>> import spooq2.transformer as T
>>> import spooq2.loader as      L
>>>
>>> pipeline = Pipeline()
>>>
>>> pipeline.set_extractor(E.JSONExtractor(
>>>     input_path="tests/data/schema_v1/sequenceFiles"
>>> ))
>>> pipeline.add_transformers([T.Mapper([
>>>     ("id",              "id",                    "IntegerType"),
>>>     ("forename",        "attributes.first_name", "StringType"),
>>>     ("surname",         "attributes.last_name",  "StringType"),
>>>     ("created_at",      "meta.created_at_ms",
↪  "timestamp_ms_to_s")])])
>>> pipeline.set_loader(L.HiveLoader(db_name="users_and_friends",
↪  table_name="users"))
>>>
>>> pipeline.execute()
```

Objects of `PipelineFactory` help data engineers and data scientists to construct and define a `Pipeline` for *Spooq*. Code Block 4.5 shows the syntax on how to use a `PipelineFactory` instance to read raw JSON data, transform, and finally persist it to a Hive table as well as an example on how to fetch a DataFrame for further processing in an ad hoc situation. The only parameters set are the type of entity to extract, the date (which could also be derived from the current day), and a time range of the loaded data. The main difference between those two examples is the lack of a `batch_size` attribute for the latter, which determines the pipeline type as ad hoc and consequently skips the loader while returning the DataFrame directly.

Code Block 4.5.: Example: PipelineFactory

```
>>> pipeline_factory = PipelineFactory()
>>>
>>> #  Load user data partition with applied mapping, filtering,
>>> #  and cleaning transformers to a hive database. (ETL use case)
>>> pipeline_factory.execute({
>>>       "entity_type": "user",
>>>       "date": "2018-10-20",
>>>       "batch_size": "daily"})
>>>
>>> #  Fetch user dataset with applied mapping, filtering,
>>> #  and cleaning transformers. (ELT (ad hoc) use case)
>>> df = pipeline_factory.execute({
>>>       "entity_type": "user",
>>>       "date": "2018-10-20",
>>>       "time_range": "last_day"})
```

Instances of `PipelineFactory` provide three public methods. Fetching metadata from an external production system is triggered by `get_metadata()`. The method `get_pipeline()` returns a ready-to-be-executed `Pipeline` instance. It can be used if a user wants to validate or adapt an automatically generated `Pipeline`. For maximum autonomy, `execute()` is provided. This method calls internally `get_pipeline()` and executes the received object.

The exemplary expert system used by the `PipelineFactory` class is shown in Section 4.2.8 to serve as a reference implementation. It demonstrates how to obtain relevant information to construct and execute a *Spooq* pipeline by providing a few input variables. All necessary parameters are derived with the help of a rule-based expert system called *spooq_rules*.

4.2.3. Extractors

The primary purpose of an extractor is to fetch data and return it as a PySpark DataFrame. All extractor implementations inherit from the superclass `Extractor`, which is located in the subpackage `extractor`, as shown in Figure 4.17. At the time of writing this thesis, two general extractor classes are implemented, which serve multiple use cases.

Figure 4.17.: Class Diagram: Spooq's Extractor Subpackage

4.2.3.1. JSONExtractor

This class supports the extraction of JSON data from sequence and text files. The necessary variable `input_path` can be directly set or derived from the `base_path` and `partition` attributes. Code Block 4.6 provides two examples to showcase an implicit and an explicit definition of the `input_path`. PySpark's built-in method for parsing JSON files (`spark.sparkContext.read.json()`) is used internally for the conversion to DataFrames. Besides, the `input_path` variable

is stripped of *"hdfs://"* prefixes and ensures that the path ends with *"/*"*.

Code Block 4.6.: Example: JSONExtractor

```
>>> from spooq2 import extractor as E

>>> extractor =
↪  E.JSONExtractor(input_path="tests/data/schema_v1/sequenceFiles")
>>> extractor.input_path == "tests/data/schema_v1/sequenceFiles" + "/*"
True

>>> extractor = E.JSONExtractor(
>>>     base_path="tests/data/schema_v1/sequenceFiles",
>>>     partition="20200201"
>>> )
>>> extractor.input_path == "tests/data/schema_v1/sequenceFiles" +
↪  "/20/02/01" + "/*"
True
```

4.2.3.2. JDBCExtractor

The `JDBCExtractor` class fetches data from relational databases through a JDBC connection instead of extracting data from a file-based source. The built-in function `pyspark.sql.DataFrameReader.jdbc` from PySpark is used in the background for the data extraction. A `cache` flag can be set to avoid re-fetching from the source database due to the lazy evaluation of Spark's DataFrames.

Loading a complete table from an external database is supported by the `JDBCExtractorFullLoad` class. The user has to define a query and parameters for the JDBC connection. Code Block 4.7 shows an exemplary application of this class.

Code Block 4.7.: Example: JDBCExtractorFullLoad

```
>>> from spooq2 import extractor as E
>>>
>>> extractor = E.JDBCExtractorFullLoad(
>>>     query="""
>>>         select id, first_name, last_name, gender, created_at test_db
>>>         from users""",
>>>     jdbc_options={
>>>         "url": "jdbc:postgresql://localhost/test_db",
>>>         "driver": "org.postgresql.Driver",
```

```
>>>         "user": "read_only",
>>>         "password": "test123",
>>>     },
>>> )
>>>
>>> df = extractor.extract()
```

To incrementally load data from a table, `JDBCExtractorIncremental` can be used. In order to keep track of already extracted data, the class uses a persisted log table defined by the parameters `table`, `db`, and `partition_column`, all prefixed with `spooq2_values_`. Combing the information from the log table with the provided `partition` attribute allows avoiding to load redundant records. The main requirement for this is a column in the source table, which increases every time a specific record is altered, in most cases provided by a timestamp when the row was updated. This column is used to set lower and upper limits based on previously logged extractions. Code Block 4.8 defines an extractor to incrementally fetch new users based on the `updated_at` column (default value).

Code Block 4.8.: Example: JDBCExtractorIncremental

```
>>> import spooq2.extractor as E
>>>
>>> # Boundaries derived from previously logged extractions
>>> # => ("2020-01-31 03:29:59", False)
>>>
>>> extractor = E.JDBCExtractorIncremental(
>>>     partition="20200201",
>>>     jdbc_options={
>>>         "url": "jdbc:postgresql://localhost/test_db",
>>>         "driver": "org.postgresql.Driver",
>>>         "user": "read_only",
>>>         "password": "test123",
>>>     },
>>>     source_table="users",
>>>     spooq2_values_table="spooq2_jdbc_log_users",
>>> )
>>>
>>> extractor._construct_query_for_partition(extractor.partition)
'select * from users where updated_at > "2020-01-31 03:29:59"'
>>>
>>> df = extractor.extract()
```

4.2.4. Transformers

`Transformers` are used for the alteration of data and contain the most (business) logic. Figure 4.18 presents a similar picture as for the `extractor` subpackage. All classes inherit from a single `Transformer` superclass. The presented five subclasses of `Extractor` cover basic use cases for ETL and ELT processes. They can be generally split into two categories, *filtering* and *restructuring*.

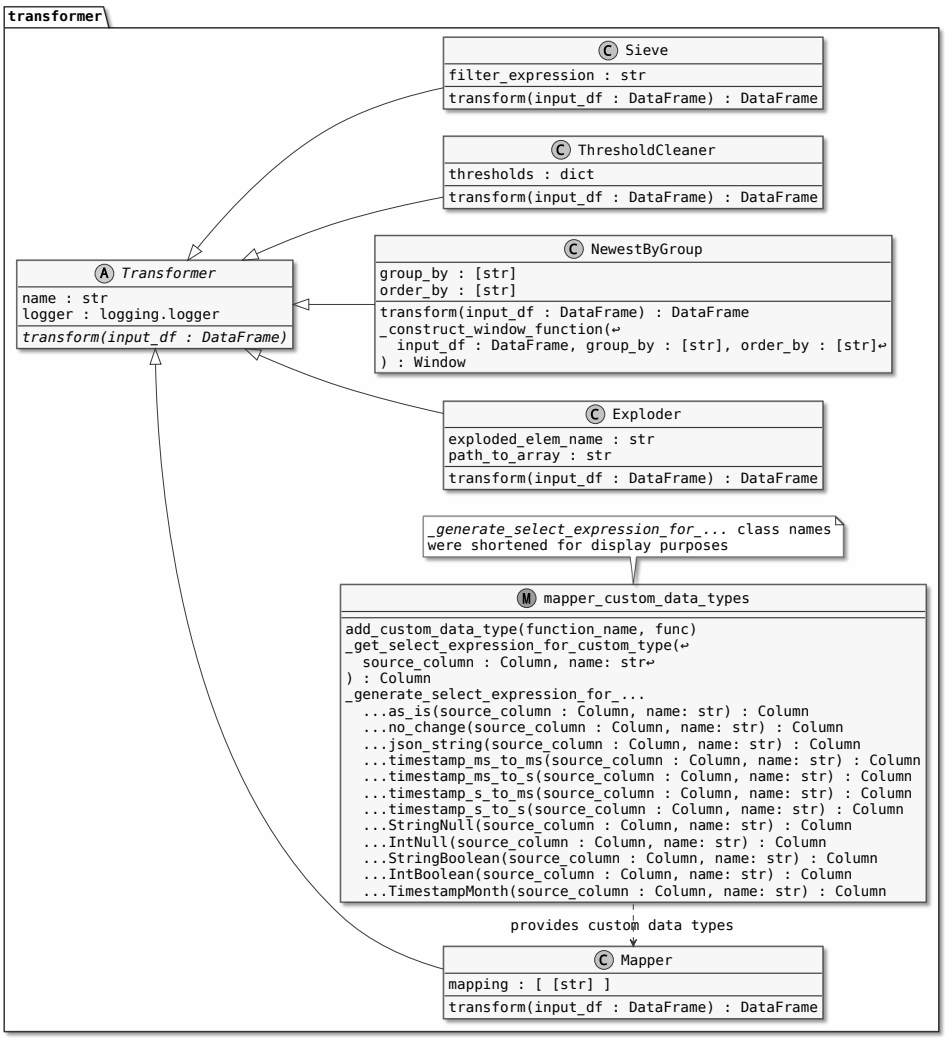


Figure 4.18.: Class Diagram: Spooq's Transformer Subpackage

4.2.4.1. Filtering

The `Sieve` transformer applies a filter expression on a DataFrame, which can consist of any valid Spark SQL code. Possible expressions can vary from regex, like `last_name rlike "^.{7}$"`, to simple string comparisons, like `gender = "f"`.

Instead of filtering on record level, `ThresholdCleaner` instances work on cell level. A dictionary (`thresholds`) defines lower and upper limits on column level, with a default value if a cell contains a number outside said limits. Setting the size of a person to *NULL* if he or she is not between 70 and 250 centimeters tall would be expressed as: `"size_cm": {"min": 70, "max": 250}`, skipping the default value. Multiple column ranges can be set, although, only columns containing numerical values are supported.

The third transformer used for filtering is based on groups of records. `NewestByGroup` solves the problem of event data where a batch can contain multiple records concerning the same entity. If a object is updated multiple times, multiple update event messages will be produced and therefore ingested into a data lake. For further processing, only the most-up-to-date message per id is of interest (or allowed), which is why older information about an id should be discarded. The `NewestByGroup` transformer filters an incoming dataset by applying the function `row_number` from the `spark.sql.function` module on a window, defined by the `group_by` and `order_by` parameters, and selects the first row.

4.2.4.2. Restructuring

One of the most visible transformations on a dataset is probably the restructuring and mapping to a different table schema. Especially for hierarchical data sources, like JSON, flattening nested attributes makes further processing easier and increases the comprehensibility.

PySpark supports next to primitive data types also *complex* data types. `StructTypes` define nested hierarchies of key/value pairs, similar to nested Python's dictionaries. Columns of type `MapType` act similar to `StructType` columns, but without hierarchical structures. Columns with `ArrayType` as data type contain an arbitrary number of elements with the same data type, similar to a list in Python. Useful data is sometimes only stored as an appendix to an entity within an arrays. Katz et al. (2020) specify that "resource objects that are related to the primary data and/or each other ('included resources')" have to be of type array to comply with the JSON-API version 1.0.

The `Exploder` transformer uses PySpark's built-in method for exploding array-based columns (`spark.sql.functions.explode()`), which potentially increases the total number of rows. The source column is defined by `path_to_array` and the target column by `exploded_elem_name`. This transformer is often combined with the `Sieve` class to filter out exploded rows which are of no interest.

The `Mapper` transformer represents the most complicated and unique transformer of all currently implemented. Its sole parameter contains information about the desired output schema, supported by definitions of the source columns and target data types. This `mapping` parameter consists of a list of tuples with one tuple per output column. The following three items can fully define the resulting table schema on column level:

Target Column Name (`name`)
: This attribute sets the name of the column in the resulting output DataFrame.

Source Column Name/Path (`source_column`)
: Points to the source column in the input DataFrame. If the source value is part of a struct, it will point to the path of the actual value. For example: *data.relationships.sample.data.id*, where id is the desired value.

Data Type (`data_type`)
: Data types can be built-in PySpark data types, predefined custom data types, or injected, ad hoc data types.

Code Block 4.9.: Example: Mapping Parameter for Mapper Class

```
>>> from spooq2 import transformer as T
>>>
>>> mapping = [
>>>     ('id',          'data.relationships.food.data.id',
    ↪  'StringType'),
>>>     ('updated_at',  'elem.attributes.updated_at',
    ↪  'timestamp_ms_to_s'),
>>>     ('deleted_at',  'elem.attributes.deleted_at',
    ↪  'timestamp_ms_to_s'),
>>>     ('names',        'elem.attributes.name',             'array')
>>> ]
>>> transformer = T.Mapper(mapping=mapping)
>>> df = transformer.transform(input_df)
```

Code Block 4.9 gives an example of a simple mapping for the `Mapper` transformer. The transformed DataFrame will have a schema with four columns, namely *id*, *updated_at*, *deleted_at*, and *names*. The *id* values are taken from a struct column in the input DataFrame accessed through the path *data.relationships.food.data.id* and cast as a string. *updated_at*'s and *deleted_at*'s data type is a custom data type, defined in the `mapper_custom_data_types` module, which converts Unix timestamp in milliseconds to seconds, while removing outliers. The last column contains a list of names which will be returned without any casting.

Custom data types can include any logic which is expressible as PySpark SQL or via Python UDFs (User Defined Functions). Currently following custom data types are available:

`as_is` (keep, `no_change`, and `without_casting` as aliases)
: Returns the source column without any processing or casting.

`json_string`
: Converts a (complex) column to its JSON equivalent.

`timestamp_ms_to_ms`
: Removes outliers of Unix timestamp in milliseconds.

`timestamp_ms_to_s`
: Converts a Unix timestamp from milliseconds to seconds and removes outliers.

timestamp_s_to_s
Removes outliers of Unix timestamp in seconds.

timestamp_s_to_ms
Converts a Unix timestamp from seconds to milliseconds and removes outliers.

StringNull
Discards any values and casts *NULL* as string. Useful for GDPR related anonymization.

IntNull
Discards any values and casts *NULL* as integer. Useful for GDPR related anonymization.

StringBoolean
Returns *1* as a string if the source contains valid content. Useful for GDPR related anonymization.

IntBoolean
Returns *1* as an integer if the source contains valid content. Useful for GDPR related anonymization.

TimestampMonth
Sets a timestamp to the first day of its given month. Useful for GDPR related anonymization of birthdays.

Figure 4.19 outlines the logical flow of each mapping tuple to construct a global DataFrame select-expression. The `Mapper` transformer applies said select-expression and returns the input DataFrame in the given schema. The first step checks whether the value of `data_type` is a built-in of Spark or a custom data type. A data type will be interpreted as a PySpark built-in if it is a member of `pyspark.sql.types`. The method `_get_select_expression_for_custom_type()` from the `mapper_custom_data_types` module will be called if it is not an importable PySpark data type. The method for custom data types returns a PySpark SQL expression, chosen and parameterized by `name`, `source_column`, and `data_type`. In case of a built-in Spark data

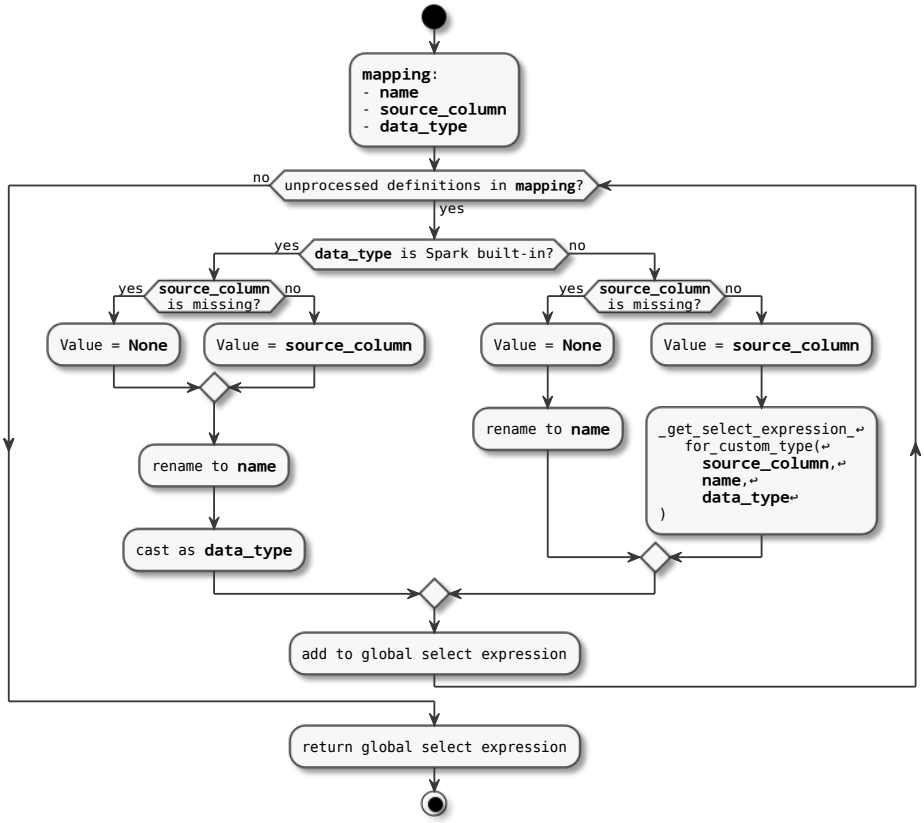


Figure 4.19.: Activity Diagram: Constructing Select Statement With the Mapper Transformer

type, the column `source_column` will be renamed to `name` and cast as `data_type`. If `source_column` is missing in the incoming DataFrame, the value will be set to *NULL*, for both built-in and custom data types. The responsibility of constructing select-expression-construction is forwarded to the `mapper_custom_data_types` module for data types that are not found within PySpark.

The `mapper_custom_data_types` module supports adding custom data types in runtime via the `add_custom_data_type()` method.

Injecting a custom UDF-based data type defined within the data pipeline is shown in Code Block 4.10.

Code Block 4.10.: Example: Adding Custom Data Type in Runtime

```
>>> import spooq2.transformer.mapper_custom_data_types as custom_types
>>> import spooq2.transformer as T
>>> from pyspark.sql import Row, functions as F, types as sql_types
>>>
>>> def hello_world(source_column, name):
>>>     "A UDF (User Defined Function) in Python"
>>>     def _to_hello_world(col):
>>>         if not col:
>>>             return None
>>>         else:
>>>             return "Hello World"
>>>
>>>     udf_hello_world = F.udf(_to_hello_world, sql_types.StringType())
>>>     return udf_hello_world(source_column).alias(name)
>>>
>>> input_df = spark.createDataFrame(
>>>     [Row(hello_from=u'tsivorn1@who.int'),
>>>      Row(hello_from=u''),
>>>      Row(hello_from=u'gisaksen4@skype.com')]
>>> )
>>>
>>> custom_types.add_custom_data_type(function_name="hello_world",
    ↪  func=hello_world)
>>> transformer = T.Mapper(mapping=[("hello_who", "hello_from",
    ↪  "hello_world")])
>>> df = transformer.transform(input_df)
>>> df.show()
+-----------+
|  hello_who|
+-----------+
|Hello World|
|       null|
|Hello World|
+-----------+
```

4.2.5. Loaders

Loaders accept a single PySpark DataFrame and persist it to a target system. Implementations of specific loaders inherit from the superclass `Loader` as shown in Figure 4.20. To support an additional database, file storage, or file format, only a specific loader class is needed. All subsequent data transformations happen on PySpark DataFrames and are therefore agnostic to the output format. Currently, only loading into a Hive table is implemented.

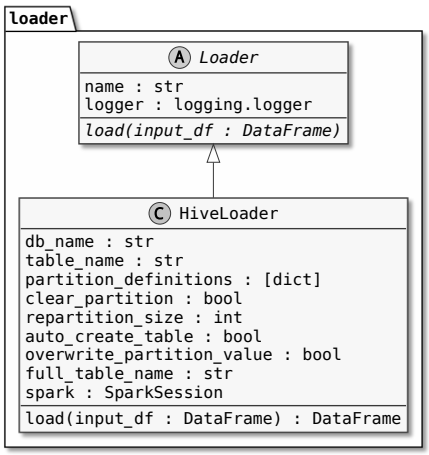


Figure 4.20.: Class Diagram: Spooq's Loader Subpackage

4.2.5.1. Hive Loader

The `HiveLoader` class provides an interface to persist DataFrames to Hive tables through PySpark's `pyspark.sql.DataFrameWriter` class. It can be used for incremental loads with the help of partitions or for full loads on table level.

Code Block 4.11 gives an example for a partitioned and a non-partitioned target table. The `full_loader` instance takes a DataFrame, reduces the number of output files (`repartition_size`) and creates or overwrites the target table defined in `table_name`, and `db_name`. Line 10 shows how to define an `incremental_loader` object that partitions the dataset according to `partition_definitions` and inserts the DataFrame into a Hive table partition.

Code Block 4.11.: Example: Hiveloaders for Incremental and Full Loads

```
1  >>> import spooq2.extractor as E
2  >>>
3  >>> full_loader = L.HiveLoader(
4  >>>     db_name="users_and_friends",
5  >>>     table_name="friends_mapping",
6  >>>     auto_create_table=True,
7  >>>     repartition_size=5
8  >>> )
9  >>>
```

```
>>> incremental_loader = L.HiveLoader(
>>>     db_name="users_and_friends",
>>>     table_name="friends_mapping",
>>>     partition_definitions=[
>>>         {"column_name": "dt",
>>>          "column_type": "IntegerType",
>>>          "default_value": 20200201}
>>>     ],
>>>     clear_partition=True,
>>>     overwrite_partition_value=True,
>>>     repartition_size=40
>>> )
```

Figure 4.21 showcases the complex logic of `HiveLoader` to cover multiple use cases. Repartitioning is performed with the default or given value to keep the number of files at a reasonable size. If partitioning is requested via `partition_definitions`, the existence of columns, their values, and respective data types are asserted to be valid or corrected if necessary. The attribute `overwrite_partition_value` determines if the existing partition values of the relevant partitioning column should be used or overwritten by a provided default value. Validations are made to ensure that the DataFrame shares the same structure as an existing target table, as SQL relies on the column sequence instead of their names. The `clear_partition` flag can be set to drop any existing partitions from the target table, to enable back-filling (reloading of already processed batches). If the defined target does not yet exist, `HiveLoader` applies potential partition configuration on the DataFrame and writes the complete dataset to the table specified by `full_table_name`.

4.2.6. Tests

One of *Spooq's* quality criteria is to have all relevant code unit tested. All extractors, transformers, and loaders are checked for syntactical, logical, and data qualitative errors. This does, however, not render integration tests of explicit pipelines with specific datasets redundant. Composing an ETL pipeline for frequent processing tasks of a distinct entity type should always come with specific integration tests

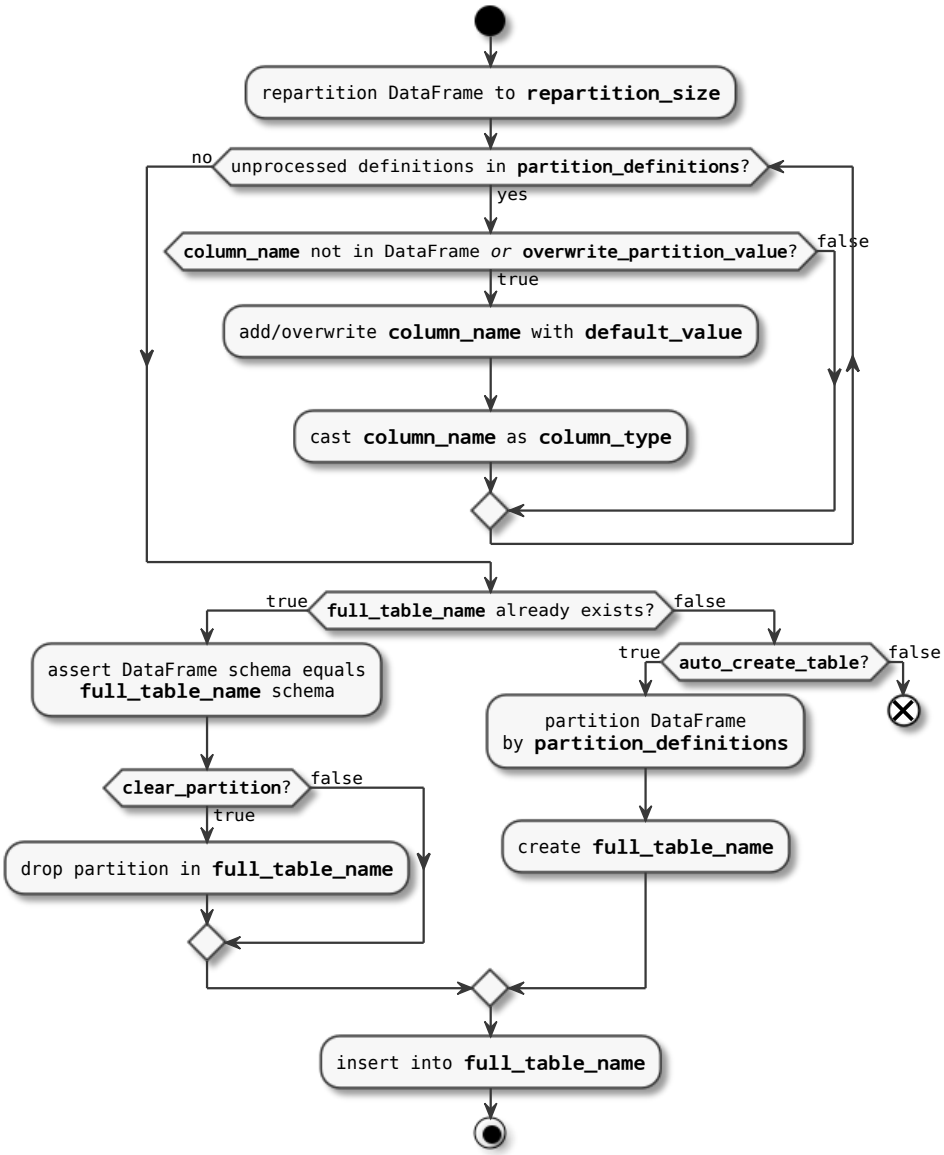


Figure 4.21.: Activity Diagram: Loading Into a Hive Table

to ensure continuous correctness in combination with *Spooq's* unit tests.

The test suite of *Spooq* is implemented with the *pytest* framework, developed by Oliveira (2018). In combination with *pipenv's* environments, calling **`pytest unit`** from the test directory is enough to run all implemented tests. *pytest* supports over 315 plugins of which *Spooq* currently uses eleven. The most interesting to describe here are *html*, *random-order*, *cov*, *ipdb*, and *pytest-spark*. *html* generates reports in HTML format for the test results which allows for easy spotting of failed tests and faster debugging. *random-order* shuffles the execution order of the tests to uncover dependencies between individual tests. Unit tests should, by definition, be independent of each other. *cov* generates a report similar to *html*, but for code coverage. It shows which lines of code have not been executed by running tests. This can lead to uncovering gaps in the test coverage or give a quantitative measurement for the completeness of tests. However, 100 percent test coverage can give the false impression that the application under test is free of bugs, which is a dangerous conclusion. Another *pytest* helper to mention is called *ipdb*. Strictly speaking, *ipdb* is an interactive debugger but can be used for inspecting Python test methods as well. The most important plugin is *pytest-spark* which provides a local PySpark instance to be used by *pytest*.

All testing relevant code and data is stored in the *tests* directory to separate its content from the actual application logic. The subfolder *unit* contains the python files which describe the test cases. Another mention-worthy directory is *data* which stores a set of test data in different formats which is used by numerous test cases.

Code Block 4.12 provides an example of a typical unit test. Methods decorated with `@pytest.fixture` provide reusable objects which are costly or complicated to create. Most of the unit tests have a fixture called `input_df`, `default_params`, and sometimes also a `default_transformer` to keep the test's structure similar and quicker to comprehend. The function `test_count()` compares the number of records between the implemented transformation and

an explicitly exploded DataFrame with PySpark's built-in function. The second test, named `test_exploded_array_is_added()`, checks if the exploded elements are correctly added to the schema. The last test method, called `test_array_is_converted_to_struct()`, finally validates that an array was successfully converted to a struct with the help of converting PySpark DataFrames to Python dictionaries.

Code Block 4.12.: Example: Unit Tests for Exploder Transformer

```
import pytest
import json
from pyspark.sql import functions as sql_funcs

from spooq2.transformer import Exploder


class TestExploding(object):
    @pytest.fixture(scope="module")
    def input_df(self, spark_session):
        return spark_session.read.parquet("data/schema_v1/parquetFiles")

    @pytest.fixture()
    def default_params(self):
        return {"path_to_array": "attributes.friends",
        ↪  "exploded_elem_name": "friend"}

    def test_count(self, input_df, default_params):
        expected_count = input_df.select(
            sql_funcs.explode(input_df[default_params["path_to_array"]])
        ).count()
        actual_count =
        ↪  Exploder(**default_params).transform(input_df).count()
        assert expected_count == actual_count

    def test_exploded_array_is_added(self, input_df, default_params):
        transformer = Exploder(**default_params)
        expected_columns = set(
            input_df.columns + [default_params["exploded_elem_name"]]
        )
        actual_columns = set(transformer.transform(input_df).columns)

        assert expected_columns == actual_columns

    def test_array_is_converted_to_struct(self, input_df, default_params):
        def get_data_type_of_column(df, path=["attributes"]):
            record = df.first().asDict(recursive=True)
            for p in path:
                record = record[p]
                return type(record)

        current_data_type_friend = get_data_type_of_column(input_df,
        ↪  path=["attributes", "friends"])
        assert issubclass(current_data_type_friend, list)

```

```
43          transformed_df = Exploder(**default_params).transform(input_df)
44          transformed_data_type = get_data_type_of_column(transformed_df,
            ↪  path=["friend"])
45
46          assert issubclass(transformed_data_type, dict)
```

Code Block 4.13 demonstrates the output for the tests defined in Code Block 4.12 by running the command **`pytest`** in a shell session.

Code Block 4.13.: Example: Results of Exploder Transformer Unit Tests

```
1  ============================= test session starts
   ↪  ==============================
2  platform linux2 -- Python 2.7.18, pytest-3.10.1, py-1.8.1, pluggy-0.13.1
3  Using --random-order-bucket=module
4  Using --random-order-seed=346005
5
6  Spark will be initialized with options:
7    spark.app.name: spooq-pyspark-tests
8    spark.default.parallelism: 1
9    spark.driver.extraClassPath: ../bin/custom_jars/sqlite-jdbc.jar
10   spark.dynamicAllocation.enabled: false
11   spark.executor.cores: 1
12   spark.executor.extraClassPath: ../bin/custom_jars/sqlite-jdbc.jar
13   spark.executor.instances: 7
14   spark.io.compression.codec: lz4
15   spark.rdd.compress: false
16   spark.shuffle.compress: false
17   spark.sql.shuffle.partitions: 1
18 rootdir: /home/david/projects/spooq2/tests, inifile: pytest.ini
19 plugins: doubles-1.5.0, sugar-0.9.2, cov-2.5.1, random-order-0.8.0,
   ↪  metadata-1.8.0, env-0.6.2, assume-1.2.1, mock-2.0.0, html-1.19.0,
   ↪  pspec-0.0.3, spark-0.5.2
20 collected 6 items
   ↪
21
22 unit/transformer/test_exploder.py
   ↪
23 Exploding
24  ✓ exploded array is added
25  ✓ count
26
27 Mapper for Exploding Arrays
28  ✓ name is set
29  ✓ str representation is correct
30
31 Exploding
32  ✓ array is converted to struct
33
34 Mapper for Exploding Arrays
35  ✓ logger should be accessible
36
37
38 ========================== 6 passed in 7.20 seconds
   ↪  ==========================
```

Details about the amount of tests, their code coverage, and how to write tests for new components will be provided in the evaluation sections 5.3.3 and 5.3.4.1.

4.2.7. Documentation

Next to testing, documentation plays an important role for *Spooq's* maintainability. Each implemented component of the ETL process, including the classes they interact with, should be described and explained in Python's docstrings. Documented code can be viewed directly from a Python shell or a notebook via the docstrings. Additional tools allow to generate a self-contained documentation as a web page or a PDF document. Please see the appendix in Section "Appendix A: Spooq Documentation" for the included PDF version of *Spooq's* documentation.

Spooq uses *Sphinx* to create its HTML and PDF documentation. Brandl (2019), the creator of *Sphinx*, describes it as "... a tool that makes it easy to create intelligent and beautiful documentation". It uses textual descriptions of modules, classes, and functions within the source code. Parameters, attributes, and hierarchies are independently parsed and added to the documentation. Combining the extracted data with additional information provided via *reStructuredText* files allows for the automatic generation of *Spooq's* documentation.

Sphinx supports numerous extensions of which *Spooq* currently uses eight. Most importantly to mention are *napoleon*, *intersphinx*, and *PlantUML*. The plugin *napoleon* provides support for docstrings in the style of Google and NumPy, which are often easier to read in the source code. Linking to external documentations, like Spark's online documentation, is enabled by *intersphinx*. Creating diagrams and graphs is supported by *PlantUML*, which was also used to create the activity, class, and data flow diagrams in this thesis. It is a Java library for creating vector-based graphs from textual scripts. Those scripts can be included in documents with *reStructuredText* to easily keep the included diagrams up-to-date and version-controlled.

Code Block 4.14 gives an example of the `Exploder` documentation within its docstring. Python docstrings are delimited by three double quotes directly after the declaration of an object. The extension *napoleon* provides several sections which are being interpreted differently by *Sphinx* and result therefore in individual styles of display. The `Examples` header instructs *Sphinx* to parse the following text as code and apply code highlighting. `Parameters` list input variables with their names, data types, and descriptions. `Warnings` and `Notes` are means to emphasize distinct parts of the text.

Code Block 4.14.: Example: Docstring of Exploder Transformer

```
Explodes an array within a DataFrame and
drops the column containing the source array.

Examples
--------
>>> transformer = Exploder(
>>>     path_to_array="attributes.friends",
>>>     exploded_elem_name="friend",
>>> )

Parameters
----------
path_to_array : :any:`str`, (Defaults to 'included')
    Defines the Column Name / Path to the Array.
    Dropping nested columns is not supported.
    Although, you can still explode them.

exploded_elem_name : :any:`str`, (Defaults to 'elem')
    Defines the column name the exploded column will get.
    This is important to know how to access the Field afterwards.
    Writing nested columns is not supported.
    The output column has to be first level.

Warning
-------
**Support for nested column:**

path_to_array:
    PySpark cannot drop a field within a struct. This means the specific
    ↪ field
    can be referenced and therefore exploded, but not dropped.
exploded_elem_name:
    If you (re)name a column in the dot notation, is creates a first level
    ↪ column,
    just with a dot its name. To create a struct with the column as a
    ↪ field
    you have to redefine the structure or use a UDF.

Note
----
The :meth:`~spark.sql.functions.explode` method of Spark is used
↪ internally.
```

```
39
40  Note
41  ----
42  The size of the resulting DataFrame is not guaranteed to be
43  equal to the Input DataFrame!
```

Figure 4.22 shows the HTML output for the example defined in Code Block 4.14, generated by *Sphinx* with the *Read the Docs Sphinx Theme*.

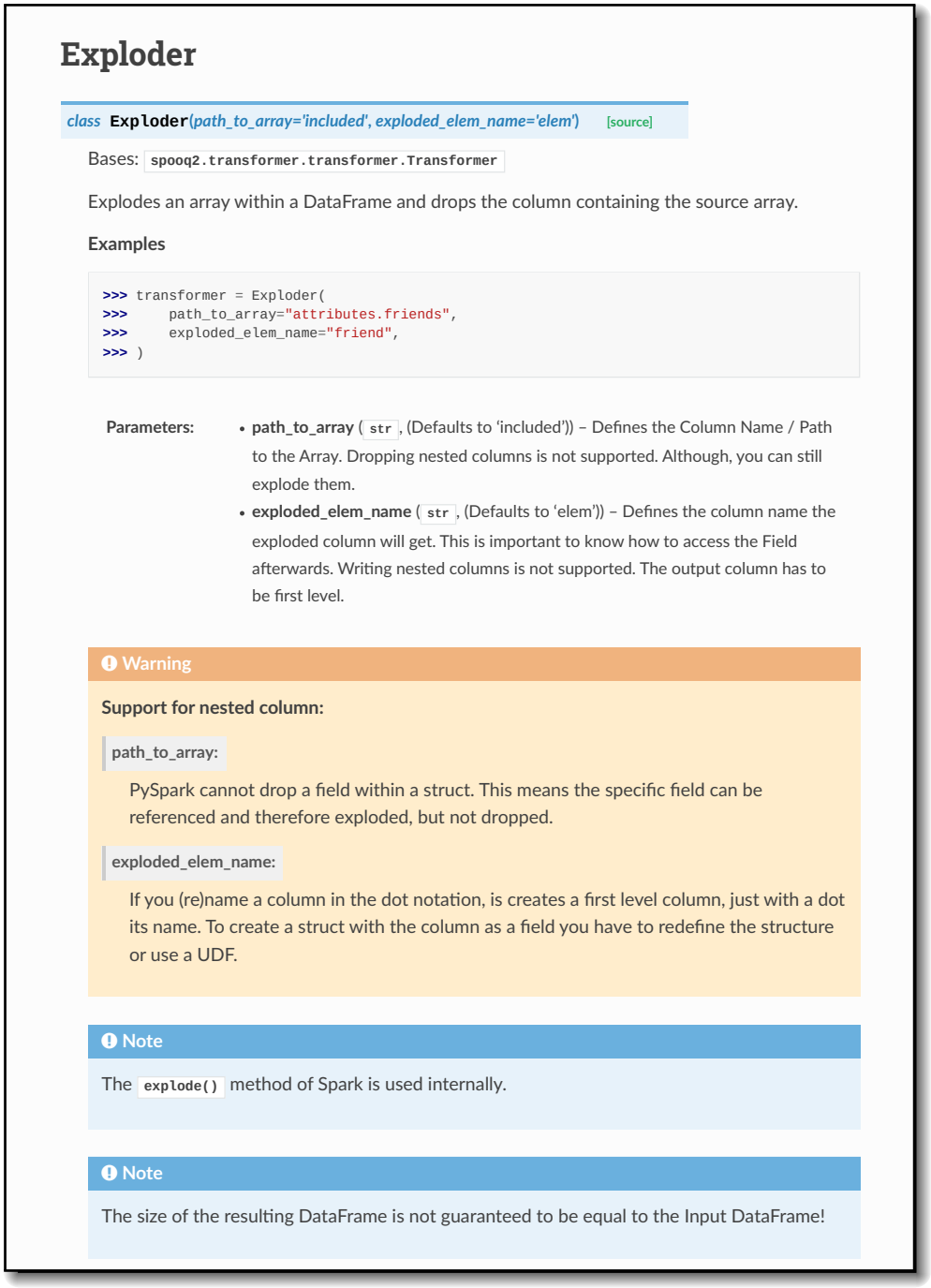
4.2.8. Semi-Automatic Configuration by Reasoning

The design of a data pipeline depends mainly on the domain expertise of a data engineer and the context of its concrete use case. Outsourcing domain knowledge to an expert system and defining a vocabulary of context attributes allows to hand the construction of ETL or ELT processes over to an automated service. The `PipelineFactory` class represents an interface of *Spooq* which passes context variables to an external system, receives configuration parameters, and builds the desired pipeline.

For the sake of demonstration, an exemplary expert system was implemented, called *spooq_rules*. It consists mainly of a REST endpoint and an expert system in the background. Python's *Flask* library serves the HTTP interface. The expert system is implemented with *Experta* which relies on a knowledge base of production rules. The repository of metadata is implemented in pure Python.

4.2.8.1. Inference

spooq_rules makes use of the production rule reasoning engine of *Experta*. It divides the domain of *Spooq* pipelines into smaller subdomains to have a clear separation of the respective knowledge bases. There are `KnowledgeEngine` classes (knowledge bases) for the enrich-



Exploder

class **Exploder**(*path_to_array='included', exploded_elem_name='elem'*) [source]

Bases: `spooq2.transformer.transformer.Transformer`

Explodes an array within a DataFrame and drops the column containing the source array.

Examples

```
>>> transformer = Exploder(
>>>     path_to_array="attributes.friends",
>>>     exploded_elem_name="friend",
>>> )
```

Parameters:

- **path_to_array** (`str` , (Defaults to 'included')) – Defines the Column Name / Path to the Array. Dropping nested columns is not supported. Although, you can still explode them.
- **exploded_elem_name** (`str` , (Defaults to 'elem')) – Defines the column name the exploded column will get. This is important to know how to access the Field afterwards. Writing nested columns is not supported. The output column has to be first level.

Warning

Support for nested column:

path_to_array:

PySpark cannot drop a field within a struct. This means the specific field can be referenced and therefore exploded, but not dropped.

exploded_elem_name:

If you (re)name a column in the dot notation, is creates a first level column, just with a dot its name. To create a struct with the column as a field you have to redefine the structure or use a UDF.

Note

The `explode()` method of Spark is used internally.

Note

The size of the resulting DataFrame is not guaranteed to be equal to the Input DataFrame!

Figure 4.22.: Example: HTML Documentation of Exploder Transformer

ment of context variables, for determining the ETL component types, and for inferring the appropriate initialization parameters.

spooq_rules currently features 14 *Spooq*-relevant `KnowledgeEngines` which are used for tasks in different sub-domains. The following list gives an overview:

`Context`: Enriches initial context variables

`ExtractorName`: Determines the appropriate extractor class

- **`JSONExtractor`**: Deducts the necessary parameters to initialize a `JSONExtractor` class

`TransformerNames`: Determines the appropriate transformer classes

- **`Exploder`**: Deducts the necessary parameters to initialize an `Exploder` class
- **`Sieve`**: Deducts the necessary parameters to initialize a `Sieve` class
- **`Mapper`**: Deducts the necessary parameters to initialize a `Mapper` class
  - **`Column`**: Deducts the necessary attributes for a single mapping used by the `Mapper` class
- **`ThresholdCleaner`**: Deducts the necessary parameters to initialize a `ThresholdCleaner` class
- **`NewestByGroup`**: Deducts the necessary parameters to initialize a `NewestByGroup` class

`LoaderName`: Determines the appropriate loader class

- **`HiveLoader`**: Deducts the necessary parameters to initialize a `HiveLoader` class
  - **`PartitionDefinition`**: Deducts the necessary attributes for a single partition definition used by the `HiveLoader` class
- **`ByPass`**: Responds with an empty object to keep the internal process structure consistent

The context rule engine (`Context`) takes care of enriching the initial set of context variables received from *Spooq's* `PipelineFactory`. Code Block 4.15 shows a shortened version of the `Context` class. Structure, general flow, and style of rules are quite similar to other `KnowledgeEngines` of *spooq_rules*, which makes the presentation of the `Context` class representative for them as well. The remainder of this section will therefore focus on the `Context` knowledge base. A more complete example will be shown in the demonstration part of this thesis.

All `KnowledgeEngines` are initialized with a `response` attribute containing an empty dictionary or an error message denoting that no rules were fired. After resetting the engine, a single fact is constructed from the available attributes and declared on the `KnowledgeEngine's` instance. This fact potentially satisfies the condition of some rules which therefore can create new facts. The rule described by the `set_default_pipeline_type()` method from Code Block 4.15 is for example activated if neither the `pipeline_type` nor the `batch_size` is defined by any fact in the working memory. The consequent of this rule assigns the values *"ad_hoc"* and *"no"* to the Python variables `pipeline_type` and `batch_size`, respectively. Those variables are inserted into the `response` dictionary with their respective names as keys. On line 15, a fact is created with `pipeline_type` and `batch_size` as keys and the variables with the same name as values. This new fact is directly passed to the `declare()` function of the rule's class which submits the fact to the working memory. After running the engine, all relevant results from the inference process are acquired back via the `response` attribute of the engine.

Code Block 4.15.: Example: Rule Definitions for Enrichment of Context Variables in spooq_rules

```
1  class Context(KnowledgeEngine):
2
3      def __init__(self):
4          super().__init__()
5          self.response = {}
6
7      @Rule(NOT(Fact(pipeline_type=W())),
8            NOT(Fact(batch_size=W())),
9            salience=4)
```

```
    def set_default_pipeline_type(self):
        pipeline_type = "ad_hoc"
        batch_size = "no"
        self.response["pipeline_type"] = pipeline_type
        self.response["batch_size"] = batch_size
        self.declare(Fact(pipeline_type=pipeline_type,
        ↪  batch_size=batch_size))

    @Rule(NOT(Fact(level_of_detail=W())),
                Fact(pipeline_type="ad_hoc"))
    def set_level_of_detail_for_ad_hoc(self):
        level_of_detail = "all"
        self.response["level_of_detail"] = level_of_detail
        self.declare(Fact(level_of_detail=level_of_detail))

    @Rule(NOT(Fact(pipeline_type=W())),
            (Fact(batch_size="no")),
            salience=5)
    def set_pipeline_type_according_to_no_batch_size(self):
        pipeline_type = "ad_hoc"
        self.response["pipeline_type"] = pipeline_type
        self.declare(Fact(pipeline_type=pipeline_type))
```

Continuing the ad hoc pipeline example, as shown above in Code Block 4.5, only `entity_type`, `date`, and `time_range` are initially provided by an actor. Running the inference engine of *spooq_rules* with those three attributes results in four inferred context variables as shown in Code Block 4.16.

Code Block 4.16.: Example: Context Variables Query from spooq_rules

```
>>> import requests
>>> import json
>>>
>>> initial_variables = {
>>>     "entity_type": "user",
>>>     "date": "2018-10-20",
>>>     "time_range": "last_day"
>>> }
>>> context_variables = requests.post("http://localhost:5000/context/get",
>>>                                   json=initial_variables).json()
>>>
>>> print(json.dumps(context_variables, indent=2))
{
"pipeline_type": "ad_hoc",
"entity_type": "user",
"level_of_detail": "all",
"level_of_detail_int": 10,
"batch_size": "no",
"time_range": "last_day",
"date": "2018-10-20"
}
```

Code Block 4.17 gives an idea about the inference procedure by examining the output logs. It shows the activation, ordering, and execution of rules. The rule `set_default_pipeline_type` was triggered first as the given fact satisfied its LHS and because of its raised priority (`salience`). A new fact with `pipeline_type` and `batch_size` attributes was declared within the RHS of said rule, which resulted in the activation and firing of rule `set_level_of_detail_for_ad_hoc`. The inference process stopped as no more rules were applicable, and therefore the agenda was blank.

Code Block 4.17.: Example: Context Variables Inference by spooq_rules

```
INFO:werkzeug: * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
DEBUG:experta.watchers.AGENDA:0: 'set_default_pipeline_type' '<f-0>'
INFO:experta.watchers.RULES:FIRE 1 set_default_pipeline_type: <f-0>
INFO:experta.watchers.FACTS: ==> <f-2>: Fact(pipeline_type='ad_hoc',
↪  batch_size='no')
INFO:experta.watchers.ACTIVATIONS: <== 'set_default_pipeline_type': <f-0>
↪  [EXECUTED]
INFO:experta.watchers.ACTIVATIONS: ==> 'set_level_of_detail_for_ad_hoc':
↪  <f-2>, <f-0>
DEBUG:experta.watchers.AGENDA:0: 'set_level_of_detail_for_ad_hoc' '<f-2>,
↪  <f-0>'
INFO:experta.watchers.RULES:FIRE 2 set_level_of_detail_for_ad_hoc: <f-2>,
↪  <f-0>
INFO:experta.watchers.FACTS: ==> <f-3>: Fact(level_of_detail='all')
INFO:experta.watchers.ACTIVATIONS: <== 'set_level_of_detail_for_ad_hoc':
↪  <f-0>, <f-2> [EXECUTED]
INFO:experta.watchers.ACTIVATIONS: ==> 'set_integer_to_level_of_detail':
↪  <f-3>
DEBUG:experta.watchers.AGENDA:0: 'set_integer_to_level_of_detail' '<f-3>'
INFO:experta.watchers.RULES:FIRE 3 set_integer_to_level_of_detail: <f-3>
INFO:experta.watchers.FACTS: ==> <f-4>: Fact(level_of_detail_int=10)
INFO:werkzeug:127.0.0.1 - - [12/Mar/2020 18:06:28] "POST /context/get
↪  HTTP/1.1" 200 -
```

4.2.8.2. API

The endpoint of *spooq_rules* provides several routes, all accepting POST requests. */pipeline/get* serves as the main entry point as it implicitly calls all necessary functions internally and responds with a JSON object which contains all information needed by `PipelineFactory`. For specific use cases and debugging, the internal routes */<xxx>/name* and */<xxx>/params/<yyy>* can be accessed directly. *xxx* stands for extractor, transformer, or loader while *yyy* has to be substituted by the actual class names. The */context/get* route is called internally

and infers pipeline information about the general context which is subsequently needed for other `KnowledgeEngines`. This context information is used to fetch additional metadata for the current use case.

Defining a strict structure of *spooq_rules'* response object ensures compatibility with *Spooq's* `PipelineFactory`. The inference service responds with a JSON object that contains at least the keys `extractor`, `transformers`, and `loader`. Each of these keys corresponds to an object which itself contains a name and another object with parameters. In the case of transformers, an array needs to be supplied rather than a single object.

Table 4.1 shows an example for the JSON schema with the names and parameters of the components.

```
{
    "extractor": {
        "name": "Type1Extractor",
        "params": {"key 1": "val 1", "key N": "val N"}
    },
    "transformers": [
        {
            "name": "Type1Transformer",
            "params": {"key 1": "val 1", "key N": "val N"}
        },
        {
            "name": "Type2Transformer",
            "params": {"key 1": "val 1", "key N": "val N"}
        },
        {
            "name": "Type3Transformer",
            "params": {"key 1": "val 1", "key N": "val N"}
        }
    ],
    "loader": {
        "name": "Type1Loader",
        "params": {"key 1": "val 1", "key N": "val N"}
    }
}
```

Table 4.1.: Exemplary Input Data for PipelineFactory

5. Demonstration and Evaluation

The following sections demonstrate the functionality of *Spooq* based on practical examples. The dataset utilized by the examples will be introduced, and its format, content, and context explained. *Spooq* will be demonstrated by designing and executing a batch-oriented ETL processes, as well as an ad hoc ELT data pipeline, both within a local stand-alone Spark deployment. The ETL example will be replicated within a Cloudera Hadoop on-premises environment, and the ELT pipeline will be repeated within a Databricks workspace on Microsoft Azure to indicate the broad applicability of the implemented library. How to extend *Spooq* will be demonstrated by exemplarily adding a new extractor, transformer, and loader class to its codebase. The last demonstration example will show the capabilities of *Spooq* to automate the construction of its pipelines with the support of an expert system. The evaluation part of this section examines the degree of achievement concerning the evaluation criteria set out in Section 3.2.3. It checks for functionality, adherence to engineering principles, decrease in complexity, and increase in quality.

5.1. Running Example

This section introduces the reader to the context of the examples utilized for demonstration purposes of *Spooq* and *spooq_rules*. The choice of the JSON format for the input data is explained. Syntax and visualization of the data are shown and the used example dataset is described in more detail.

5.1.1. Format of Input Data

This thesis will demonstrate its ETL and ELT processes by using JSON records as raw data for the extraction process. There are multiple reasons for choosing this semi-structured data format. According to Stack Overflow — a community-centric website dedicated to helping developers and programmers — questions about JSON are more numerous than XML, and YAML with 286,717, 191,430, and 8,043 questions, respectively, as of March 2020. (Stack Exchange, Inc., 2020a, 2020b, 2020c). The benefits of improved readability and flexibility are described in the following paragraphs.

Human-readable
: Due to its lesser verbosity, compared to XML, JSON-formatted data is easier to read by humans. Attribute names and values are physically close to each other, and hierarchies are directly presented. (Kleppmann, 2017)

Flexibility
: In comparison to CSV or other simpler formats, the JSON format is able to express almost any possible complex data structure. The data types string, numeric types, (JSON-)object, array, boolean, and null allow for nested and hierarchical schemata. This is especially relevant for event-based records, which often include related data of interest in arrays. (Droettboom, 2019)

5.1.2. Syntax and Format of Processing Steps

This thesis uses the following conventions to demonstrate different aspects of a running example.

Input Data: Table 5.1 showcases exemplary input data with light gray background and syntax highlighting, which is surrounded by a top and bottom rule.

```
{
    "data": [
        {
            "type": "input data example",
            "id": "1"
        },
        {
            "type": "another example row",
            "id": "2"
        }
    ]
}
```

Table 5.1.: Exemplary Input Data

Output Data Results are presented in left-aligned tables with a black, monospace font and white background. The header row sits between a bold top and a plain bottom rule and is typeset with bold font as shown in Table 5.2.

type	id
input data example	1
another example row	2
another example row	3
another example row	n

Table 5.2.: Exemplary Output Data

5.1.3. Type of Output Data

The scope of this thesis is to propose a solution to extract, transform, and load data in a format that can directly be used by data scientists or by data warehouse applications. Therefore, the requirements for loaded data are:

SQL compatibility

A cell must only contain a single value. No complex data types like arrays or structs should be used. This keeps querying from, and loading to, a relational database system via JDBC/ODBC as compatible as possible. Not all DBMS handle complex data types the same nor do all JDBC/ODBC drivers support them out-of-the-box without any workarounds. Almost all production-proven data warehouse implementations take SQL as a common base language. This makes them compatible with the output of *Spooq*. Normalization of the tables is not part of this scope. ELT pipelines make the exception, as their output is used dynamically within exploratory analyses, and will, therefore, not be loaded to a target database at this point.

Deduplication

Only one object per id and partition is allowed. In the case of event-based ingestion, only the most up-to-date record per batch is to be selected. Allowing an object id in multiple partitions provides a historical view on past data. Its preciseness is directly influenced by the batch size of the input data and the frequency of *Spooq's* scheduling.

Cleansing

Impossible and wrong values are removed. Consistency for formats of single entities is provided. For example, converting all timestamps to UTC Unix timestamps in milliseconds.

Anonymization

As the output data is accessible by different stakeholders, data-protective measurements have to be taken. Almost all use cases for data exploration, insight generation, and reporting rely on aggregated information. This allows for early anonymization of personal data without losing relevant information, for example, first name, last name, or phone number.

5.1.4. Example Dataset

Yelp Inc. provides a dataset to use for data-related projects of academic nature. It consists of multiple JSON files, that total to more than eight gigabytes of uncompressed content. After accepting Yelp's license, it can be downloaded from https://www.yelp.com/dataset/download.

The dataset of Yelp provides six text files that contain a single JSON representation per line. The following list gives an overview of each entity type and their description according to the documentation by Yelp Inc (2020):

user.json
: Contains information about registered Yelp users.

business.json
: Includes data about businesses which are connected with Yelp.

review.json
: Represents reviews done by users about businesses with their respective ids.

checkin.json
: Shows when and how often a business was checked-in on.

tip.json
: Similar to reviews but contains shorter messages by users.

photo.json
: Links photos for businesses with captions and categories.

For the demonstration of *Spooq*, *user* and *business* records provide the most interesting data. The *user* dataset contains explode-able arrays which corresponds well to event-based messages and their array-based linked data. As a second entity type, *business* fits well as it consists of nested data structures, often found on semi-structured data.

5.1.4.1. User Entity Type

Table 5.3 shows an adapted user example, taken from the dataset documentation provided via Yelp Inc (2020). Sebastien, with the user-id *Ha3iJu77CxlrFm-vQRs_8g*, registered on the first of January in 2011 and became part of Yelp's elite team in the years 2012 and 2013. He has three friends and 1,032 fans. In his time as an active Yelp member, he wrote 56 reviews, which were rated with an average of 4.31. He voted other reviews and comments as useful, funny, and cool 21, 88, and 15 times, respectively. Attributes prefixed with `compliment_` count the amount of compliments Sebastien received by other users over time. For the sake of demonstration purposes, three columns (`p_year`, `p_month`, and `p_day`) were derived from `yelping_since` to support easier partitioning.

5.1.4.2. Business Entity Type

A second example taken from the dataset's documentation by Yelp Inc (2020) illustrates a business entity, as shown in Table 5.4. Garaje is an open Mexican restaurant with the id of *tnhfDv5Il8E-aGSXZGiuQGg*, located in San Francisco. Its precise place is described by its address attributes and its geospatial latitude/longitude coordinates. 1,198 reviews resulted so far in an average rating of 4.5. The `attributes` object specifies its parking and take-out capabilities (not all possible attributes are listed). Types of served food are noted in the array `categories`. The attribute `hours` contains opening hours per weekday, stored as strings in a JSON object. To support easier partitioning, three columns (`p_year`, `p_month`, and `p_day`) were added to the dataset, filled with random values.

A script was created to preprocess Yelp's datasets. It splits comma separated strings into JSON-compliant arrays, adds `p_year`, `p_month`, and `p_day` attribute for partitioning, and stores the records dynamically into dedicated subfolders regarding the partition values. Please

```
{
    "user_id": "Ha3iJu77CxlrFm-vQRs_8g",
    "name": "Sebastien",
    "review_count": 56,
    "yelping_since": "2011-01-01",
    "friends": [
        "wqoXYLWmpkEH0YvTmHBsJQ",
        "KUXLLiJGrjtSsapmxmpvTA",
        "6e9rJKQC3n0RSKyHLViL-Q"
    ],
    "useful": 21,
    "funny": 88,
    "cool": 15,
    "fans": 1032,
    "elite": [
        2012,
        2013
    ],
    "average_stars": 4.31,
    "compliment_hot": 339,
    "compliment_more": 668,
    "compliment_profile": 42,
    "compliment_cute": 62,
    "compliment_list": 37,
    "compliment_note": 356,
    "compliment_plain": 68,
    "compliment_cool": 91,
    "compliment_funny": 99,
    "compliment_writer": 95,
    "compliment_photos": 50,
    "p_year": "2011",
    "p_month": "01",
    "p_day": "01",
}
```


Table 5.3.: Example of Yelp's User Type (Yelp Inc, 2020)

```
{
    "business_id": "tnhfDv5Il8E-aGSXZGiuQGg",
    "name": "Garaje",
    "address": "475 3rd St",
    "city": "San Francisco",
    "state": "CA",
    "postal code": "94107",
    "latitude": 37.7817529521,
    "longitude": -122.39612197,
    "stars": 4.5,
    "review_count": 1198,
    "is_open": 1,
    "attributes": {
        "RestaurantsTakeOut": true,
        "BusinessParking": {
            "garage": false,
            "street": true,
            "validated": false,
            "lot": false,
            "valet": false
        },
    },
    "categories": [
        "Mexican",
        "Burgers",
        "Gastropubs"
    ],
    "hours": {
        "Monday": "10:00-21:00",
        "Tuesday": "10:00-21:00",
        "Friday": "10:00-21:00",
        "Wednesday": "10:00-21:00",
        "Thursday": "10:00-21:00",
        "Sunday": "11:00-18:00",
        "Saturday": "10:00-21:00"
    },
    "p_year": "2017",
    "p_month": "12",
    "p_day": "15",
}
```


Table 5.4.: Example of Yelp's Business Type (Yelp Inc, 2020)

refer to Section “Appendix B: Preparation of Yelp’s Raw Data for Examples” for the source code.

This section described the input datasets for the subsequent demonstration of *Spooq*. The syntax and visualization of data presentation were defined. SQL compliance, deduplication, cleansing, and anonymization were listed and described as requirements for transformed data. Records of type *user* and *business* from the current Yelp JSON dataset are being used, which were presented here in more detail. The reasons for the choice of this dataset were its proximity to practical business data, JSON format, and permissive data agreement for academic purposes.

The demonstration of *Spooq*, which uses examples based on the dataset specified above, follows in the next section.

5.2. Demonstration

The demonstration part is divided into two distinct use cases, one for batch-oriented ETL pipelines, and one for ad hoc ELT processing flows. The flexibility of *Spooq* is demonstrated by the execution of a data pipeline on different Spark distributions in varying environments. Adding new components showcases the evolvability and required effort for new data sources, transformations, and sinks for *Spooq*. The last section showcases the capabilities of automation through reasoning. *Spooq_rules* represents an exemplary expert system designed to dynamically build *Spooq* data pipelines with the help of context variables, metadata, and a rule-based inference engine.

5.2.1. ETL Batch Application

A common use case for incremental data processing is to load newly registered users into a database. For this demonstration, a frequency

of daily batches was chosen. An unrelated streaming service fetches and writes new users, partitioned by reception date, into following folder structure: *user/p_year=<year>/p_month=<month>/p_day=<day>*. Every 24 hours, a scheduler executes an ETL batch job, which processes all new entries from the past day.

Code Block 5.1 shows the corresponding data pipeline, implemented with *Spooq*. It features one extractor, six transformation steps, and a loader component. The primary purpose of the resulting dataset is to enable analyses on users who are of importance for social interaction on the Yelp platform. To be of importance, a user has to have an average rating of at least 2.5 stars and is connected with other users. The only parameter at runtime is the date, which will be passed as shown in Code Block 5.2. All remaining configuration is defined within the script and does not change for subsequent executions.

Code Block 5.1.: ETL Demonstration: Defining Pipeline Manually

```
from datetime import datetime
import sys
import os

from spooq2.pipeline import Pipeline
import spooq2.extractor as   E
import spooq2.transformer as T
import spooq2.loader as      L


def run(batch_date):

    pipeline = Pipeline()

    pipeline.set_extractor(E.JSONExtractor(
        input_path=os.path.join("user",
            datetime.strftime(batch_date,
            ↪  "p_year=%Y/p_month=%m/p_day=%d"))
        )
    )

    pipeline.add_transformers([
        T.Exploder(exploded_elem_name="friends_element",
        ↪  path_to_array="friends"),
        T.ThresholdCleaner(thresholds={"average_stars": {"max": 5, "min":
        ↪  1}}),
        T.Sieve(filter_expression="""
            isnotnull(friends_element) and
            friends_element <> \"None\"""),
        T.Sieve(filter_expression="average_stars >= 2.5"),
        T.Mapper(mapping=[("user_id",       "user_id",
        ↪  "StringType"),
```

```
                            ("review_count",  "review_count",
                            ↪  "LongType"),
                            ("average_stars", "average_stars",
                            ↪  "DoubleType"),
                            ("elite_years",   "elite",
                            ↪  "json_string"),
                            ("friend",        "friends_element",
                            ↪  "StringType")]),
            T.NewestByGroup(group_by=["user_id", "friend"],
            ↪  order_by=["review_count"])
        ])

        pipeline.set_loader(L.HiveLoader(
                auto_create_table=True,
                partition_definitions=[
                    {"default_value": batch_date.year,
                        "column_type": "IntegerType",
                        "column_name": "p_year"},
                    {"default_value": batch_date.month,
                        "column_type": "IntegerType",
                        "column_name": "p_month"},
                    {"default_value": batch_date.day,
                        "column_type": "IntegerType",
                        "column_name": "p_day"}],
                overwrite_partition_value=True,
                repartition_size=10,
                clear_partition=True,
                db_name="user",
                table_name="users_daily_partitions_scripted"
        ))

        pipeline.execute()


    if __name__ == "__main__":
        date_arg = sys.argv[1]
        batch_date = datetime.strptime(date_arg, "%Y-%m-%d")
        run(batch_date)
```

Code Block 5.2.: ETL Demonstration: Executing Pipeline Manually

```
$ python etl_pipeline_user.py "2018-10-20"
```

The workflow of the ETL pipeline printed in Code Block 5.1 reads as follows:

1. Raw data, in the format of JSON files, is read from the respective input directory. This limits the extraction to the specific day, which was passed as a parameter. The `JSONExtractor` also takes care of input path sanitization, text decoding, and conversion of the JSON records to a PySpark DataFrame.

2. Due to the requirement of SQL compatibility, as described in Section 5.1.3, arrays are not recommended for the output table and have, therefore, to be exploded. The `Exploder` transformer explodes the column `friends` into `friends_element`, which consequently multiplies a user record by the number of its friends.

3. The first `Sieve` transformer drops users who do not have any friends.

4. A `ThresholdCleaner` instance searches for outliers in the attribute named `average_stars` and sets all values which are not between *1.0* and *5.0* to *None*.

5. Continuing with a cleansed `average_stars` column, a second `Sieve` transformer filters out all users with an average star rating below *2.5*, as defined for users of interest with respect to social interaction.

6. As not all columns of the source data are relevant to store in a data warehouse, only a subsection of them is selected. The `user_id` attribute is needed to identify a user. The attributes `review_count` and `average_stars` are an indicator of the user's level of activity on the platform. Column `elite_years` stores the years in which the status of important actors on Yelp was already assigned to the user. For this attribute, a string data type was chosen, which contains a JSON compatible value.

7. The last transformer groups all records by `user_id` and `friend` columns and sorts by `review_count`. Selecting only records that have the highest `review_count` filters out out-of-date rows and deduplicates the data.

8. The `HiveLoader` takes the transformed DataFrame and persists it to the `users_daily_partitions` table in the Hive database `user`. For consistency and performance reasons, the partitioning structure from the input directories is kept.

Table 5.5 outputs five exemplary rows of data, loaded to the Hive table. Each record represents a user -> friend connection with attrib-

utes describing the user. Consequently, multiple rows per user are expected if they have more than one friend. The last three columns are for partitioning and can substantially decrease querying time if used properly.

user_id	review_ count	average_ stars	elite_ years	friend	p_ year	p_ month	p_ day
NPkNHqoqi6r-rWvVgkctEgA	1	5.0	[]	MZtEfYn7Zax-2vm5KqeYF9A	2018	10	20
1kfReHhvBPe-JR4byU22m8g	1	5.0	[]	B70GUlNqjiA-WMRp23QdD2A	2018	10	20
1kfReHhvBPe-JR4byU22m8g	1	5.0	[]	kT5WbT2KUwt-A8uboNpKNoA	2018	10	20
rp7RdcyBuFk-eZiYaEyLDOg	15	4.27	[]	eTiHUbr5K6j-6gQzM0qdj8g	2018	10	20
OeiXd6eRICd-lqq00K4sldQ	2	3.0	[]	7L55hN0BaVe-xaROqXUgj3A	2018	10	20

Table 5.5.: ETL Demonstration: Pipeline Output

5.2.2. ELT Ad Hoc Use Case

Data scientists and business analysts sometimes need more information about a given entity type than what is stored in a data warehouse. Machine learning assisted data products rely on a multitude of attributes and values. Their effect on the accuracy is often not evident in foresight. Instead of persisting every available data characteristic to data warehouse tables, querying the source on an ad hoc basis satisfies this use case and avoids redundancy as well as wasted disk space. Data lakes' principle of schema-on-read allows for re-processing of data from its earliest state. This process is furthermore referred to as ELT (Extract, Load, and Transform).

An ELT pipeline is demonstrated in this section to showcase the functionality of *Spooq* for ad hoc use cases. Business entities from the Yelp dataset provide an interesting source of information that can be used to construct machine learning models, for example, a prediction model for star ratings based on attributes of a business. Identically to the `user` entity type, the incoming, raw *business*

data is stored in partitioned directories following the structure: *business/p_year=<year>/p_month=<month>/p_day=<day>*. An ELT *Spooq* pipeline for interactive purposes is shown in Code Block 5.3.

Code Block 5.3.: ELT Demonstration: Defining and Executing Pipeline Manually

```
>>> import datetime
>>> import sys
>>> import os
>>>
>>> from spooq2.pipeline import Pipeline
>>> import spooq2.extractor as   E
>>> import spooq2.transformer as T
>>> import spooq2.loader as      L
>>>
>>> pipeline = Pipeline()
>>> date = datetime.datetime.strptime("2018-10-20", "%Y-%m-%d")
>>>
>>> input_paths = []
>>> for delta in range(0,7):
>>>     day = date - datetime.timedelta(delta)
>>>     partition_path = datetime.datetime.strftime(
>>>         day, "p_year=%Y/p_month=%m/p_day=%d"
>>>     )
>>>     input_paths.append(os.path.join("business", partition_path))
>>>
>>>
↪  pipeline.set_extractor(E.JSONExtractor(input_path=",".join(input_paths)))
>>>
>>> mapping = [
>>>     ( "business_id",       "business_id",     "StringType"  ),
>>>     ( "name",              "name",            "StringType"  ),
>>>     ( "address",           "address",         "StringType"  ),
>>>     ( "city",              "city",            "StringType"  ),
>>>     ( "state",             "state",           "StringType"  ),
>>>     ( "postal_code",       "postal_code",     "StringType"  ),
>>>     ( "latitude",          "latitude",        "DoubleType"  ),
>>>     ( "longitude",         "longitude",       "DoubleType"  ),
>>>     ( "stars",             "stars",           "LongType"    ),
>>>     ( "review_count",      "review_count",    "LongType"    ),
>>>     ( "categories",        "categories",      "json_string" ),
>>>     ( "open_on_monday",    "hours.Monday",    "StringType"  ),
>>>     ( "open_on_tuesday",   "hours.Tuesday",   "StringType"  ),
>>>     ( "open_on_wednesday", "hours.Wednesday", "StringType"  ),
>>>     ( "open_on_thursday",  "hours.Thursday",  "StringType"  ),
>>>     ( "open_on_friday",    "hours.Friday",    "StringType"  ),
>>>     ( "open_on_saturday",  "hours.Saturday",  "StringType"  ),
>>>     ( "attributes",        "attributes",      "json_string" ),
>>> ]
>>>
>>> pipeline.add_transformers([
>>>     T.Mapper(mapping=mapping),
>>>     T.ThresholdCleaner(thresholds={
>>>         "stars": {"min": 1, "max": 5},
>>>         "latitude": {"min": -90.0, "max": 90.0},
>>>         "longitude": {"min": -180.0, "max": 180.0}
>>>     }),
>>> ])
```

```
>>>
>>> pipeline.bypass_loader = True
>>>
>>> df = pipeline.execute()
```

Running the ELT pipeline portrayed in Code Block 5.3 returns the extracted and transformed DataFrame. The flow of action is described in the following enumeration:

1. A date value is given as a string scalar and parsed as a `datetime` object.

2. In addition to the given date, also data from the six precedent days is needed. `JSONExtractor` does not provide the logic of time ranges for input data, as it is solely file-based. The "`for delta in range(0,7)`" loop traverses through each day of last week and joins the appropriate input paths for the extractor.

3. Raw data is read from the respective input paths. Although the input path parameter is a single string, a comma-separated list of paths is interpreted as a set of input directories. The `JSONExtractor` takes care of input path sanitization, text decoding, and the conversion of the JSON records to a PySpark DataFrame.

4. The `Mapper` transformer takes care about the schema of the output DataFrame. The source columns `categories` (array type) and `attributes` (struct type) are transformed into JSON strings. This allows a data engineer to perform string operations on those columns and easily export or pipe them to other services, while keeping the flexibility of complex data types.

5. A `ThresholdCleaner` instance searches for outliers in the `stars` attribute and sets all values which are not between *1.0* and *5.0* to *None*. The attributes `longitude` and `latitude` are filtered similarly. The physical nature of those coordinates allows for hard limits, which inevitably only drop impossible values.

6. Instead of persisting the transformed DataFrame into a database, it is stored in the variable `df`. The person who executes the code can work with the resulting DataFrame, which contains business records from a week, cleansed by non-destructive methods, and mapped to the desired schema.

Code Block 5.4 lists the schema of the returned DataFrame, as shown by PySpark's `printSchema()` function.

Code Block 5.4.: ELT Demonstration: Pipeline Output Schema

```
>>> df.printSchema()
root
 |-- business_id: string (nullable = true)
 |-- name: string (nullable = true)
 |-- address: string (nullable = true)
 |-- city: string (nullable = true)
 |-- state: string (nullable = true)
 |-- postal_code: string (nullable = true)
 |-- latitude: double (nullable = true)
 |-- longitude: double (nullable = true)
 |-- stars: long (nullable = true)
 |-- review_count: long (nullable = true)
 |-- categories: string (nullable = true)
 |-- open_on_monday: string (nullable = true)
 |-- open_on_tuesday: string (nullable = true)
 |-- open_on_wednesday: string (nullable = true)
 |-- open_on_thursday: string (nullable = true)
 |-- open_on_friday: string (nullable = true)
 |-- open_on_saturday: string (nullable = true)
 |-- attributes: string (nullable = true)
```

5.2.3. Execution in Different Environments

The dependencies of *Spooq* are kept to a minimum to enable other companies and institutions to use it as well. The only requirements are a PySpark 2+ installation and the Python library *pandas*. This section is to show what it takes for other Spark distributions to run a data pipeline with *Spooq*. The ETL script from Section 5.2.1 is reused to demonstrate the application on a Hadoop distribution (Cloudera) and the ad hoc ELT script from Section 5.2.2 to illustrate dynamic data science applications in the cloud (Databricks).

5.2.3.1. Stand-Alone Spark

All runs, demonstrations, and tests of *Spooq* within this thesis are done on standalone Spark deployments unless stated otherwise. For development purposes, the Spark distribution *spark-2.4.3-bin-hadoop2.7* was used from the Apache Software Foundation website (spark.apache.org/downloads.html). Utilized Java version was *OpenJDK* in version 1.8.0 with Python 2.7.17. The systems used by the author for developing and testing were:

Work Laptop
Lenovo ThinkPad T470p (Core i7 7700HQ, 16 GB RAM)
Windows Subsystem for Linux 2 (WSL2)
Manjaro Gnome x64 Kernel 4.19

Private Laptop
Dell XPS 9560 (Core i7 7700HQ, 16 GB RAM)
Manjaro Gnome x64 Kernel 5.4

Chromebook
Lenovo Chromebook 500e (Celeron N4100, 8 GB RAM)
Crostini (Debian and Arch Linux)

Private Desktop
Intel Core i5 4570, 24 GB RAM
Manjaro Gnome x64 Kernel 5.4

As the majority of servers and clusters run on Linux kernel-based operating systems, other systems like Windows or macOS were not tested to run *Spooq*. Even Microsoft's very own cloud Azure consists of more Linux virtual machines than Windows Server installations, according to Vaughan-Nichols (2018).

5.2.3.2. Spark on Hadoop Distribution (Cloudera)

Cloudera does not support an online trial for its Hadoop distribution, which is why the demonstration of *Spooq* was done on their quickstart docker container from docs.cloudera.com/documentation/enterprise/5-16-x/topics/quickstart_docker_container.html. The latest version to download is 5.13, which features Hadoop 2.6 and a Spark 1.6 installation. Due to the outdated operating system of the docker container and the rather old distribution of Cloudera, some limitations had to be accepted to enable *Spooq* to work successfully:

Spark 2 Installation
: Although Cloudera 5.x does not ship with Spark 2 out of the box, it provides the framework via an explicitly configurable parcel. Due to the outdated Java version of the docker container, Spark 2.1 was the most recent, supported version. Spark versions starting from 2.2 require Java 1.8 as a runtime environment.

Python
: Due to the old distribution of CentOS within the docker container, only Python 2.6 was available. *Python-PIP* and *NumPy* had to be manually downgraded to be able to install the *pandas* library. After importing *pandas*, *Spooq* could successfully be installed directly from the source code via **`python setup.py install`**.

Missing Spark Function in Spark 2.1
: *Spooq* uses the method `desc_nulls_last()` of PySpark's SQL functions for its sorting within the `NewestByGroup` transformer. Using the default descending sorting order (`desc()`) results in null values preceding the highest values and therefore increases the risk of severe data quality issues. Consequently, for the execution of this ETL pipeline, the `NewestByGroup` transformer was removed.

Code Block 5.5 shows the changes needed for the script used in Section 5.2.1 to be executable on the Cloudera environment. Spark on

Cloudera loads files per default from HDFS, which made it necessary to explicitly point to a local file path via the *file://* prefix. As pointed out in the description of the limitations above, the `NewestByGroup` transformer was removed.

Code Block 5.5.: ETL Demonstration: Changes needed for Spark on Hadoop (Cloudera)

```
$ diff -U 1 -t etl_pipeline_user.py etl_pipeline_user_cloudera.py --color

--- etl_pipeline_user.py            2020-03-15 13:34:25.438385976 +0100
+++ etl_pipeline_user_cloudera.py   2020-03-15 13:34:09.068237647 +0100
     pipeline.set_extractor(E.JSONExtractor(
-        input_path=os.path.join("user",
+        input_path=os.path.join("file:///data/user",
             datetime.strftime(batch_date,
                 ↪  "p_year=%Y/p_month=%m/p_day=%d"))
@@ -30,3 +30,2 @@
         T.ThresholdCleaner(thresholds={"average_stars": {"max": 5, "min":
             ↪  1}}),
-        T.NewestByGroup(group_by=["user_id", "friend"],
 ↪  order_by=["review_count"])
     ])
```

Figure 5.1 shows HUE, the browser-accessible editor and viewer for Hive databases in Cloudera distributions. The displayed query outputs ten rows from the table generated by the ETL script, which is printed in Figure 5.1.

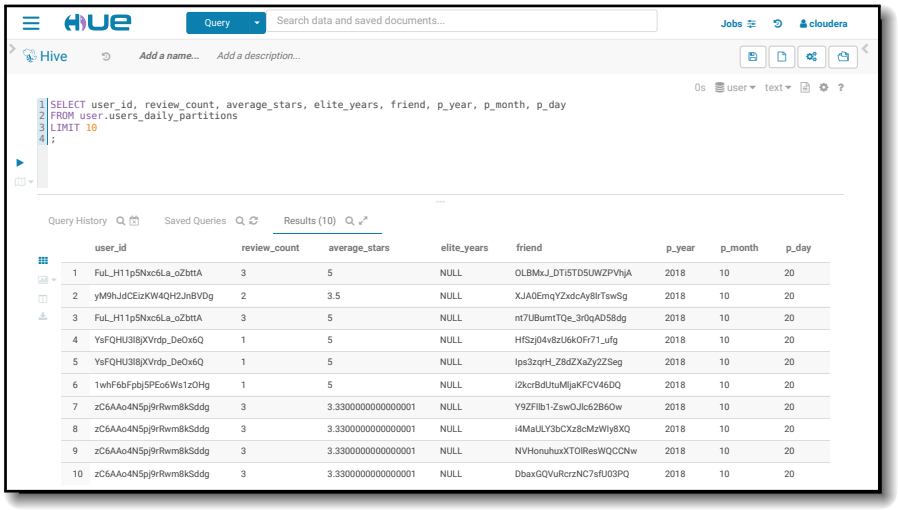


Figure 5.1.: ETL Demonstration: Querying Table Output in HUE

Please see Section “Spark on Hadoop Distribution (Cloudera)” in “Appendix C: Demonstration in Different Environments” for more figures and information.

5.2.3.3. Spark Cloud Distribution (Databricks)

The driving force behind the creation of Apache Spark, Matei Zaharia, co-founded Databricks in 2013. Databricks’ homepage states that the company’s Spark-based product provides a “Unified Data Analytics Platform,” which is a “. . . cloud platform for massive scale data engineering and collaborative data science.” (Databricks Inc., 2020; Zaharia, 2020)

The ELT ad hoc use case from Section 5.2.2 was tested in a Databricks workspace on Microsoft Azure to demonstrate the broad applicability of *Spooq*. A cluster was set-up with one driver and four worker nodes, operating Databricks runtime 5.5 (https://docs.databricks.com/release-notes/runtime/5.5.html). The following versions were used: Spark 2.4.3, Java 1.8.0_232, and Python 2.7.12. All nodes were running on Ubuntu 16.04.6 LTS. The only requirements for *Spooq*, Python’s *pandas* library, was already provided in version 0.19.2.

Importing *Spooq* to the Databricks cluster was straightforward and required to build an egg file locally (**`python setup.py bdist_egg`**), upload, and activate it via Databricks’ web interface. Figure 5.2 shows the successful status of importing *Spooq* to the cluster.

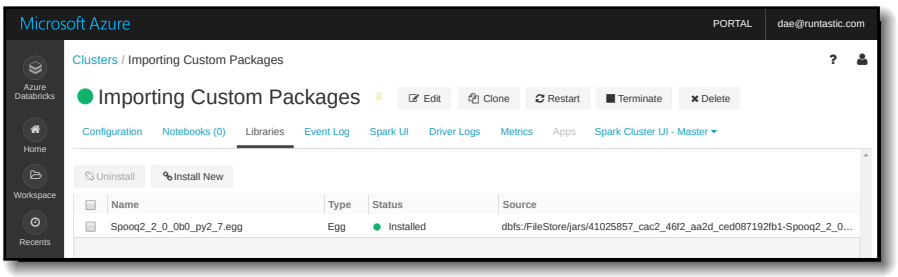


Figure 5.2.: ELT Demonstration: Importing Spooq Library into Databricks

The script demonstrated in Code Block 5.3 needed a minor adaptation, which was due to a different input path. Please see Code Block 5.6 for more details, which illustrates the difference between the original and the adapted script.

Code Block 5.6.: ELT Demonstration: Changes needed for Spark on Cloud (Databricks)

```
--- original
+++ adapted
@@ -18,3 +18,3 @@
     )
-    input_paths.append(os.path.join("business", partition_path))
+
    ↪  input_paths.append(os.path.join("/user/dae@runtastic.com/yelp_data_set",
    ↪  partition_path))
```

Figure 5.3 shows the results of *Spooq's* pipeline execution within a Databricks' notebook. The output DataFrame contains the columns and data types defined in the script. The execution details show the different Spark stages and their tasks, total execution time, and metadata like user name and time.

Figure 5.3.: ELT Demonstration: Executing Pipeline in Notebook

Please see Section “Spark Cloud Distribution (Databricks)” in “Appendix C: Demonstration in Different Environments” for more figures and information.

5.2.4. Adding New Components

Spooq's architecture was designed with evolvability in mind. This section shows one example each for adding a new extractor, transformer, and loader to its repertoire.

5.2.4.1. Adding a New Extractor

A new extractor class should inherit from the `Extractor` base class. This adds the name, string representation, and logger attributes from the superclass. As an extractor class starts by reading data and converting it to a PySpark DataFrame, a `SparkSession` object is needed to be initialized. Transformers and loaders can use the included `SparkSession` from their `input_df` DataFrame. The only mandatory thing for an extractor subclass is to override the `extract()` method, which takes no input parameters and returns a PySpark DataFrame. All configuration and parameterization should be done while initializing the class instance.

A simple example of a new extractor class is shown below in Code Block 5.7. The docstring of the class takes most space as it provides the primary information for automatic documentation. The logic of the `extract()` method is kept simple to emphasize the necessary steps for extending *Spooq*. This `CSVExtractor` implementation loads a CSV file with pre-set configuration options, like delimiter.

Code Block 5.7.: Example: Implementing a New CSV Extractor Class
src/spooq2/extractor/csv_extractor.py

```
1  from pyspark.sql import SparkSession
2
3  from extractor import Extractor
4
```

```
5  class CSVExtractor(Extractor):
6      """
7      This is a simplified example on how to implement a new extractor
   ↪   class.
8      Please take your time to write proper docstrings as they are
   ↪   automatically
9      parsed via Sphinx to build the HTML and PDF documentation.
10     Docstrings use the style of Numpy (via the napoleon plug-in).
11
12     This class uses the :meth:`pyspark.sql.DataFrameReader.csv` method
   ↪   internally.
13
14     Examples
15     --------
16     extracted_df = CSVExtractor(
17         input_file='data/input_data.csv'
18     ).extract()
19
20     Parameters
21     ----------
22     input_file: :any:`str`
23         The explicit file path for the input data set. Globbing support
   ↪   depends
24         on implementation of Spark's csv reader!
25
26     Raises
27     ------
28     :any:`exceptions.TypeError`:
29         path can be only string, list or RDD
30     """
31
32     def __init__(self, input_file):
33         super(CSVExtractor, self).__init__()
34         self.input_file = input_file
35         self.spark = SparkSession.Builder()\
36             .enableHiveSupport()\
37             .appName('spooq2.extractor: {nm}'.format(nm=self.name))\
38             .getOrCreate()
39
40     def extract(self):
41         self.logger.info('Loading Raw CSV Files from: ' + self.input_file)
42         output_df = self.spark.read.load(
43             input_file,
44             format="csv",
45             sep=";",
46             inferSchema="true",
47             header="true"
48         )
49
50         return output_df
```

A reference within *Spooq's* structure enables the shorter imports statement "`from spooq2.extractor import CSVExtractor`" instead of having to write "`from spooq2.extractor.csv_extractor import CSVExtractor`". Code Block 5.8 shows the necessary adaptations to the `__init__.py` file.

Code Block 5.8.: Example: Updating References for new CSV Extractor Class
src/spooq2/extractor/__init__.py

```
--- original
+++ adapted
@@ -1,8 +1,10 @@
 from jdbc import JDBCExtractorIncremental, JDBCExtractorFullLoad
 from json_files import JSONExtractor
+from csv_extractor import CSVExtractor

 __all__ = [
     "JDBCExtractorIncremental",
     "JDBCExtractorFullLoad",
     "JSONExtractor",
+    "CSVExtractor",
 ]
```

A test module has to be created to ensure *Spooq's* code quality. The test class `TestBasicAttributes` checks if the extractor can be initialized and inherits correctly from its superclass. Following class, `TestCSVExtraction` tests the logic of the component. In the example shown in Code Block 5.9, the count and schema of an extracted dataset is validated.

Code Block 5.9.: Example: Testing new CSV Extractor Class
tests/unit/extractor/test_csv.py

```
import pytest

from spooq2.extractor import CSVExtractor

@pytest.fixture()
def default_extractor():
    return CSVExtractor(input_path="data/input_data.csv")


class TestBasicAttributes(object):

    def test_logger_should_be_accessible(self, default_extractor):
        assert hasattr(default_extractor, "logger")

    def test_name_is_set(self, default_extractor):
        assert default_extractor.name == "CSVExtractor"

    def test_str_representation_is_correct(self, default_extractor):
        assert unicode(default_extractor) == "Extractor Object of Class
        ↪  CSVExtractor"

class TestCSVExtraction(object):

    def test_count(default_extractor):
        """Converted DataFrame has the same count as the input data"""
        expected_count = 312
        actual_count = default_extractor.extract().count()
        assert expected_count == actual_count
```

```
28
29      def test_schema(default_extractor):
30          """Converted DataFrame has the expected schema"""
31          do_some_stuff()
32          assert expected == actual
```

Generating automated documentation relies, next to docstrings in the source code, also on *rst* (reStructuredText) files. For each implemented module, a file has to be created which can contain relevant text, figures, or tables in addition to the `automodule` directive.

Code Block 5.10 displays such a file for the current example.

Code Block 5.10.: Example: Add Documentation for new CSV Extractor Class
docs/source/extractor/csv.rst

```
1  CSV Extractor
2  =============
3
4  Some text if you like...
5
6  .. automodule:: spooq2.extractor.csv_extractor
```

Adding this rst file to the table of content via an overview file is shown in Code Block 5.11.

Code Block 5.11.: Example: Updating References for new CSV Extractor Class Documentation
docs/source/extractor/overview.rst

```
1  --- original
2  +++ adapted
3  @@ -7,8 +7,9 @@
4   .. toctree::
5
6       json
7       jdbc
8  +    csv
9
10  Class Diagram of Extractor Subpackage
11  ------------------------------------------------
12  .. uml:: ../diagrams/from_thesis/class_diagram/extractors.puml
13     :caption: Class Diagram of Extractor Subpackage
```

In total, the implementation of two modules was needed. Three minor adaptations were necessary to use, test, and document the new ETL component.

5.2.4.2. Adding a New Transformer

Similar to extractors, all transformer classes should inherit from the same base class (`Transformer`). Overriding the `transform()` method is necessary, which takes a PySpark DataFrame and returns a PySpark DataFrame.

An example of a new transformer component is shown in Code Block 5.12. The `NoIdDropper` filters out records that do not have an id value set. In contrast to relational and NoSQL databases, most data lake-centric database implementations, like Hive or Databrick's Delta Lake, do not enforce unique primary keys. This transformer can be used to drop rows to ensure compatibility to databases that do not allow for non-unique keys.

Code Block 5.12.: Example: Implementing a New NoIdDropper Transformer Class
src/spooq2/transformer/no_id_dropper.py

```
1  from transformer import Transformer
2
3  class NoIdDropper(Transformer):
4      """
5      This is a simplified example on how to implement a new transformer
   ↪  class.
6      Please take your time to write proper docstrings as they are
   ↪  automatically
7      parsed via Sphinx to build the HTML and PDF documentation.
8      Docstrings use the style of Numpy (via the napoleon plug-in).
9
10     This class uses the :meth:`pyspark.sql.DataFrame.dropna` method
   ↪  internally.
11
12     Examples
13     --------
14     input_df = some_extractor_instance.extract()
15     transformed_df = NoIdDropper(
16         id_columns='user_id'
17     ).transform(input_df)
18
19     Parameters
20     ----------
21     id_columns: :any:`str` or :any:`list`
22         The name of the column containing the identifying Id values.
23         Defaults to "id"
24
25     Raises
26     ------
27     :any:`exceptions.ValueError`:
28         "how ('" + how + "') should be 'any' or 'all'"
29     :any:`exceptions.ValueError`:
```

```
30          "subset should be a list or tuple of column names"
31      """
32
33      def __init__(self, id_columns='id'):
34          super(NoIdDropper, self).__init__()
35          self.id_columns = id_columns
36
37
38      def transform(self, input_df):
39          self.logger.info("Dropping records without an Id (columns to
            ↪  consider: {col})"
40              .format(col=self.id_columns))
41          output_df = input_df.dropna(
42              how='all',
43              thresh=None,
44              subset=self.id_columns
45          )
46
47          return output_df
```

Displaying the code segments on how to add references for the new class' code and its documentation is skipped here, as it follows the same logic as with the example above for the `CSVExtractor` class. The testing module behaves similarly to the already presented example, and therefore its display is omitted. Please refer to Section "Adding NoIdDropper Transformer" in "Appendix E: Demonstration of Evolvability" for the omitted code segments.

5.2.4.3. Adding a New Loader

Loaders follow the same principles as other ETL components of *Spooq*. Having the single purpose of loading data to a persisted storage makes returning a DataFrame or other object by their `load()` method unnecessary. `load()` takes a PySpark DataFrame and returns *None*. At least the API does not force or expect anything.

Following Code Block 5.13 demonstrates a loader class that stores a DataFrame as parquet files. Due to a slightly more complex loading process, an extended docstring is provided in the code. Instances of the class `ParquetLoader` persist input DataFrames while taking care of potential partitioning definitions. Multiple assertions in the `__init__` method ensure that those partition definitions do not to contradict themselves. The `load()` method finally partitions the

DataFrame and saves it into automatically generated directories, depending on the partitioning values.

Code Block 5.13.: Example: Implementing a New Parquet Loader Class
src/spooq2/loader/parquet.py

```
from pyspark.sql import functions as F

from loader import Loader

class ParquetLoader(loader):
    """
    This is a simplified example on how to implement a new loader class.
    Please take your time to write proper docstrings as they are automatically
    parsed via Sphinx to build the HTML and PDF documentation.
    Docstrings use the style of Numpy (via the napoleon plug-in).

    This class uses the :meth:`pyspark.sql.DataFrameWriter.parquet` method internally.

    Examples
    --------
    input_df = some_extractor_instance.extract()
    output_df = some_transformer_instance.transform(input_df)
    ParquetLoader(
        path="data/parquet_files",
        partition_by="dt",
        explicit_partition_values=20200201,
        compression=""gzip""
    ).load(output_df)

    Parameters
    ----------
    path: :any:`str`
        The path to where the loader persists the output parquet files.
        If partitioning is set, this will be the base path where the partitions
        are stored.

    partition_by: :any:`str` or :any:`list` of (:any:`str`)
        The column name or names by which the output should be partitioned.
        If the partition_by parameter is set to None, no partitioning will be
        performed.
        Defaults to "dt"

    explicit_partition_values: :any:`str` or :any:`int`
                               or :any:`list` of (:any:`str` and :any:`int`)
        Only allowed if partition_by is not None.
        If explicit_partition_values is not None, the dataframe will
            * overwrite the partition_by columns values if it already exists or
            * create and fill the partition_by columns if they do not yet exist
        Defaults to None

    compression: :any:`str`
```

```
        The compression codec used for the parquet output files.
        Defaults to "snappy"


    Raises
    ------
    :any:`exceptions.AssertionError`:
        explicit_partition_values can only be used when partition_by is not None
    :any:`exceptions.AssertionError`:
        explicit_partition_values and partition_by must have the same length
    """

    def __init__(self, path, partition_by="dt", explicit_partition_values=None, compression_codec="snappy"):
        super(ParquetLoader, self).__init__()
        self.path = path
        self.partition_by = partition_by
        self.explicit_partition_values = explicit_partition_values
        self.compression_codec = compression_codec
        if explicit_partition_values is not None:
            assert (partition_by is not None,
                "explicit_partition_values can only be used when partition_by is not None")
            assert (len(partition_by) == len(explicit_partition_values),
                "explicit_partition_values and partition_by must have the same length")

    def load(self, input_df):
        self.logger.info("Persisting DataFrame as Parquet Files to " + self.path)

        if isinstance(self.explicit_partition_values, list):
            for (k, v) in zip(self.partition_by, self.explicit_partition_values):
                input_df = input_df.withColumn(k, F.lit(v))
        elif isinstance(self.explicit_partition_values, basestring):
            input_df = input_df.withColumn(self.partition_by, F.lit(self.explicit_partition_values))

        input_df.write.parquet(
            path=self.path,
            partitionBy=self.partition_by,
            compression=self.compression_codec
        )
```

Other necessary changes are shown in Section “Adding Parquet Loader” in “Appendix E: Demonstration of Evolvability”. To add the new `ParquetLoader` component to *Spooq*, the same additional adaptations as for the transformer and extractor were needed.

5.2.5. Automation Through Reasoning

Spooq supports the functionality of constructing pipelines automatically via the `PipelineFactory` class, when fed with appropriate parameters. Both examples from Section 5.2.1 and Section 5.2.2 can be automated with the help of an external service. For the proof of concept, *spooq_rules* was developed, which is described in more detail within the implementation part in Section 4.2.8. All files containing relevant code of *spooq_rules* are printed in the appendix at Section "Appendix F: Spooq Rules Source Code."

5.2.5.1. ETL Batch Application

Code Block 5.14 shows a Python console session where the ETL pipeline example from Section 5.2.1 is constructed and executed in three commands. Lines 1 and 2 import and initialize *Spooq's* `PipelineFactory`. Passing the type of entity to process, the date from when the data should be extracted, and the input batch size to the `execute()` method, as shown in lines 7 to 9, is sufficient to construct and process the data pipeline. Spark SQL's `"show tables"` command at line 4 and 11 emphasizes the creation of the new table. The last Spark SQL expression compares the table previously created by the manual script (*user.users_daily_partitions_scripted*) and the newly written table (*user.users_daily_partitions*), which proves their equality.

Code Block 5.14.: Reasoning Demonstration: ETL Pipeline

```
1  >>> from spooq2 import PipelineFactory
2  >>> pipeline_factory = PipelineFactory()
3  >>>
4  >>> spark.sql("show tables").collect()
5  [Row(database=u'user', tableName=u'users_daily_partitions_scripted',
   ↪  isTemporary=False)]
6  >>>
7  >>> pipeline_factory.execute({"entity_type": "user",
8  >>>                           "date": "2018-10-20",
9  >>>                           "batch_size": "daily"})
10 >>>
11 >>> spark.sql("show tables").collect()
```

```
[Row(database=u'user', tableName=u'users_daily_partitions',
↪  isTemporary=False),
 Row(database=u'user', tableName=u'users_daily_partitions_scripted',
↪  isTemporary=False)]
>>>
>>> spark.sql("""
>>>     select * from user.users_daily_partitions_scripted
>>>     EXCEPT
>>>     select * from user.users_daily_partitions""")
>>> ).collect()
[]
```

The logic which enables this automation comes from *spooq_rules*, which utilizes a metadata repository in the background. The log file, shown in Section "Rules Triggered by ETL Batch Pipeline Inference" in "Appendix D: Demonstration of Semi-Automatic Configuration by Reasoning," outlines the inference logic behind this ETL pipeline demonstration.

5.2.5.2. ELT Ad Hoc Use Case

Ad hoc data queries for exploration or analysis fit very well for *Spooq's* pipeline automation functionality. The spontaneous pipeline constructed in Section 5.2.2 can be expressed in a few lines of code with the help of *Spooq's* `PipelineFactory` and *spooq_rules*.

Code Block 5.15.: Reasoning Demonstration: Ad Hoc ELT Pipeline

```
>>> from spooq2 import PipelineFactory
>>> pipeline_factory = PipelineFactory()
>>>
>>> df = pipeline_factory.execute({"entity_type": "business",
>>>                                "date": "2018-10-20",
>>>                                "time_range": "last_week"})
```

Code Block 5.15 shows the necessary context variables to receive the requested DataFrame. Next to changing the type of entity, the attribute `batch_size` was switched to `time_range` compared to the previously demonstrated ETL batch application. Because the attribute `batch_size` is missing, *spooq_rules* treats the pipeline to be of interactive nature and automatically bypasses the loader to return the DataFrame directly. A log file of *spooq_rules'* reasoning process is ap-

pended at Section “Rules Triggered by ELT Ad Hoc Pipeline Inference” in “Appendix D: Demonstration of Semi-Automatic Configuration by Reasoning” for further references.

5.2.6. Summary

Two pipeline use cases were described and shown in this section. One batch-oriented ETL and one ad hoc-oriented ELT pipeline were designed and expressed as Python scripts to demonstrate the functionality of *Spooq*. The Hadoop-based Cloudera distribution was used to execute the ETL example, and the ELT data processing was demonstrated within a Databricks workspace on Microsoft Azure to show the flexibility of *Spooq* in terms of Spark distributions. Adding one extractor, transformer, and loader showed the necessary adaptations in *Spooq's* code to allow the utilization of a new ETL component, test its code, and document its usage. Three small adaptations in total were necessary to include the component to *Spooq's* package and documentation structure. The last part showcased the automation capabilities of *Spooq* with the help of an expert system-supported application, which was developed for demonstration purposes. Both examples from the first part of this section were reused to show the reduction in complexity by shifting the business logic from data engineers towards a rule-based production system.

The next section will reflect on the demonstrated functionalities and evaluate them against the criteria defined in Section 3.2.3.

5.3. Evaluation

The topic of this section is the evaluation of *Spooq's* set out evaluation criteria (EC). It checks *Spooq's* ability to fulfill big data workloads on functional and performance dimensions. The effect on complexity is reviewed in the successive section. *Spooq's* capabilities for parameterization, either manual or automatic via a reasoning service, is assessed

to evaluate its reduction in complexity. Principles of software and data engineering are given attention regarding a code-based interface, broad applicability, and evolvability. Test coverage and documentation are evaluated in the last sections with regards to *Spooq's* software quality.

5.3.1. Providing ETL Functionality for Big Data (Evaluation Category I)

The purpose of this thesis' artifact is to provide a software library that eases the process of creating data pipelines in data lakes. The evaluated criteria in this section assess the necessary capabilities and their fit for processing big data amounts.

5.3.1.1. Functionality (Evaluation Category I.1)

As described in Section 4.1.1, extraction, transformation, and loading processes are essential for data ingestion into data-centric environments. *Spooq* provides, as of writing, three extractor classes, five transformers, and one loader component as itemized in the following list:

- **Extractors (EC I.1.1)**
  - JSONExtractor
  - JDBCExtractorFullLoad
  - JDBCExtractorIncremental
- **Transformers (EC I.1.2)**
  - Exploder
  - Sieve
  - Mapper
  - ThresholdCleaner
  - NewestByGroup
- **Loaders (EC I.1.3)**
  - HiveLoader

A pipeline class (EC I.1.4) was developed to provide a linked composition for multiple ETL components.

Spooq offers multiple extractors, transformers, a loader, and a pipeline class. All evaluation criteria regarding the required ETL functionality of *Spooq* are fulfilled, as summarized in Table 5.6.

EC	Name	Status
I.1.1	Implementation of one data extractor	Fulfilled
I.1.2	Implementation of one data transformer	Fulfilled
I.1.3	Implementation of one data loader	Fulfilled
I.1.4	Implementation of one pipeline object	Fulfilled

Table 5.6.: Fulfillment of Evaluation Criteria in Category Functionality (I.1)

5.3.1.2. Scalability (Evaluation Category I.2)

Computations and processing of *Spooq* pipelines are almost exclusively performed by Spark transformations and actions. The Apache Spark framework supports and utilizes parallel computing (EC I.2.1), is designed for horizontal scaling (EC I.2.2), and sees deployment in all major cloud providers (EC I.2.3). Its processing engine is designed primarily to handle big data workloads. Executions of multiple pipelines on single, local hardware shows that *Spooq* works for non-big data as well. (Chambers & Zaharia, 2018; Karau & Warren, 2017; Ryza et al., 2017; Zaharia, 2016; Zaharia et al., 2010; Zaharia et al., 2016)

Spooq provides the capabilities to process big and non-big data. It supports parallel computing, horizontal scaling, and cloud deployments within the same scope as Apache Spark. All evaluation criteria regarding scalability of *Spooq* are therefore fulfilled, as summarized in Table 5.7.

EC	Name	Status
I.2.1	Support for parallel computing	Fulfilled
I.2.2	Support for horizontal scaling	Fulfilled
I.2.3	Compatibility with cloud services	Fulfilled

Table 5.7.: Fulfillment of Evaluation Criteria in Category Scalability (I.2)

5.3.2. Decrease Complexity of Data Pipelines (Evaluation Category II)

As shown in Section 5.2, *Spooq* hides the complexity of its processing specifics. It is parameterizable and able to automatically generate data pipelines suited for specific contexts, with the help of external services.

5.3.2.1. Parameterizable (Evaluation Category II.1)

All calls to the actual Spark functions are abstracted away to let data engineers and data scientists focus on business logic rather than on implementation details. The functionality is supported by classes that do not need to be adapted on pipeline level. All definitions are set by initialization parameters. Section 5.2.1 and Section 5.2.2 show examples of an ETL batch application (EC II.1.1) and an ad hoc ELT pipeline (EC II.1.2), respectively.

Spooq reduces the complexity of constructing data pipelines by hiding implementation details and focusing on business logic. All evaluation criteria regarding the ability to parameterize *Spooq* pipelines are fulfilled, as summarized in Table 5.8.

EC	Name	Status
II.1.1	Configure daily batch-processing pipeline via parameters	Fulfilled
II.1.2	Configure ad hoc data preparation pipeline via parameters	Fulfilled

Table 5.8.: Fulfillment of Evaluation Criteria in Category Parameterizable (II.1)

5.3.2.2. Semi-Automatic Configuration by Reasoning (Evaluation Category II.2)

An automation service of *Spooq* reduces the complexity of constructing data pipelines. The demonstration of automated pipelines, supported by *spooq_rules*, in Section 5.2.5 shows the possibilities which *Spooq* provides in this regard. Providing three context parameters suffices to build and execute the same data pipelines, as demonstrated in Sections 5.2.1 (EC II.2.1) and 5.2.2 (EC II.2.2). Necessary metadata and applied production rules are documented in "Appendix D: Demonstration of Semi-Automatic Configuration by Reasoning" at "Rules Triggered by ETL Batch Pipeline Inference" and "Rules Triggered by ELT Ad Hoc Pipeline Inference," respectively.

Spooq provides the capability to utilize expert systems to minimize the data pipeline building effort, although the reasoning itself is not part of *Spooq*. A reference implementation, independent of *Spooq*, was created for demonstration purposes, which fulfills the evaluation criteria in this category only partly, as summarized in Table 5.9.

EC	Name	Status
II.2.1	Configure daily batch-processing pipeline via reasoning	Partly fulfilled
II.2.2	Configure ad hoc data preparation pipeline via reasoning	Partly fulfilled

Table 5.9.: Fulfillment of Evaluation Criteria in Category Semi-Automatic Configuration by Reasoning (II.2)

5.3.3. Conform with Standards and Best Practices (Evaluation Category III)

Following best practices and standards eases the utilization, comprehensibility, and maintenance of software applications. Three principles of data and software engineering are evaluated in this section.

5.3.3.1. Code-Focus (Evaluation Category III.1)

Even though Spark is implemented in Scala,*Spooq* uses Python as its programming language which is undoubtedly a popular language in data-centric domains. Focusing on code (EC III.1.1) rather than on user interfaces allows for easier integration with other services, scripts, and pipelines. Section 4.2 gives an overview of *Spooq's* architecture.

Spooq is developed in Python but performs its data processing in Scala for better performance. The evaluation criterion regarding support for code-focused development of *Spooq* pipelines is fulfilled, as summarized in Table 5.10.

EC	Name	Status
III.1.1	Provide code-based interface	Fulfilled

Table 5.10.: Fulfillment of Evaluation Criteria in Category Code Focus (III.1)

5.3.3.2. Broad Applicability (Evaluation Category III.2)

All major cloud distributors provide at least one PaaS (platform as a service) product that is based on Apache Spark. Microsoft Azure cloud services provide, in addition to Databricks, Azure HDInsights, which is a big data platform utilizing Apache Spark. AWS (Amazon web services) feature a Spark deployment with their EMR (Elastic MapReduce) platform as well as compatibility with Databricks

workspaces. Google's Dataproc platform is a cloud-based service that provides Spark clusters, among other services. (Poggi, 2017)

Section 5.2.3 demonstrates several Spark environments in which *Spooq* was used. The first part describes Spark's standalone deployment on local computers (EC III.2.1). It lists different operating systems and hardware systems on which *Spooq* was developed, tested, and utilized. One example was demonstrated on a Cloudera Hadoop distribution (EC III.2.2), which utilizes Spark on Yarn. Due to the outdated operating system version of the docker container provided by Cloudera, some workarounds had to be taken. The functionality of *Spooq* was limited as well because of the outdated Spark version. Eventually, the ETL batch pipeline from Section 5.2.1 was executed successfully, and the results could be queried from the target Hive table. Databricks represents the third environment in which *Spooq* was demonstrated (EC III.2.3). Its cloud-native Apache Spark platform employs a custom deployment mode. Importing and executing of an ad hoc ELT pipeline was successful without problems.

EC	Name	Status
III.2.1	Compatibility with stand-alone Spark environment	Fulfilled
III.2.2	Compatibility with on-premises Cloudera Hadoop cluster	Fulfilled
III.2.3	Compatibility with cloud-based Databricks cluster	Fulfilled

Table 5.11.: Fulfillment of Evaluation Criteria in Category Broad Applicability (III.2)

Spooq is compatible with local stand-alone Spark environments, on-premises Hadoop clusters and cloud-based Spark distributions. All evaluation criteria regarding broad applicability of *Spooq* are fulfilled, as summarized in Table 5.11.

5.3.3.3. Evolvability (Evaluation Category III.3)

Section 5.2.4.1 demonstrates the necessary steps to implement new ETL components for *Spooq*. Adding a single class that complies with *Spooq's* API definitions is sufficient to implement a new extractor (EC III.3.1), transformer (EC III.3.2), or loader (EC III.3.3). Four additional adaptations and implementations are necessary to simplify the import syntax, provide unit tests, and integrate the code into *Spooq's* documentation. The strict definition of input and output objects ensures compatibility with other ETL components.

Spooq supports evolvability through its strict APIs of its independent ETL components. Two modules and three references are enough to implement a new ETL component that is integrated, tested, and documented. All evaluation criteria regarding the evolvability of *Spooq* are fulfilled, as summarized in Table 5.12.

EC	Name	Status
III.3.1	Add additional data extractor	Fulfilled
III.3.2	Add additional data transformer	Fulfilled
III.3.3	Add additional data loader	Fulfilled

Table 5.12.: Fulfillment of Evaluation Criteria in Category Evolvability (III.3)

5.3.4. Increase Quality of Data Pipelines (Evaluation Category IV)

Extensive tests are necessary to keep *Spooq* reliable. Documentation allows for better usability for data engineers and data scientists. As described in Section 3.2.2, tests increase the reliability, whereas documentation can help users to understand how to operate *Spooq*.

5.3.4.1. Testing (Evaluation Category IV.1)

As described in Section 4.2.6, *Spooq* emphasizes the importance of unit tests for its components. Figure 5.4 presents the code coverage of *Spooq's* test suite. The report states an average of 90 percent code coverage. No ETL component features a percentage below 75 percent (EC IV.1.1). Only a single file is needed for a newly implemented ETL component to be unit tested (EC IV.1.2), as shown in Section 5.2.4.

Coverage report: 90%

filter...

Module ↓	*statements*	*missing*	*excluded*	*coverage*
src/spooq2/__init__.py	5	0	0	100%
src/spooq2/_version.py	1	1	0	0%
src/spooq2/extractor/__init__.py	3	0	0	100%
src/spooq2/extractor/extractor.py	9	1	0	89%
src/spooq2/extractor/jdbc.py	117	28	0	76%
src/spooq2/extractor/json_files.py	41	3	0	93%
src/spooq2/extractor/tools.py	19	1	0	95%
src/spooq2/loader/__init__.py	3	0	0	100%
src/spooq2/loader/hive_loader.py	66	2	0	97%
src/spooq2/loader/loader.py	9	1	0	89%
src/spooq2/pipeline/__init__.py	3	0	0	100%
src/spooq2/pipeline/factory.py	44	3	0	93%
src/spooq2/pipeline/pipeline.py	46	5	0	89%
src/spooq2/spooq2_logger.py	29	3	0	90%
src/spooq2/transformer/__init__.py	6	0	0	100%
src/spooq2/transformer/exploder.py	9	0	0	100%
src/spooq2/transformer/mapper.py	41	0	0	100%
src/spooq2/transformer/mapper_custom_data_types.py	60	8	0	87%
src/spooq2/transformer/newest_by_group.py	24	0	0	100%
src/spooq2/transformer/sieve.py	9	1	0	89%
src/spooq2/transformer/threshold_cleaner.py	18	0	0	100%
src/spooq2/transformer/transformer.py	9	1	0	89%
Total	**571**	**58**	**0**	**90%**

coverage.py v5.0.3, created at 2020-03-21 00:08

Figure 5.4.: Spooq Code Coverage via Unit Tests

Spooq consists of classes and methods which are well tested with code coverage above 75 percent. There are no changes or adaptations needed to include test cases for additional components, except for a single module containing the actual tests. All evaluation criteria regarding the testing of *Spooq* are fulfilled, as summarized in Table 5.13.

EC	Name	Status
IV.1.1	At least 75 percent code-coverage	Fulfilled
IV.1.2	Effort to write tests for ETL components	Fulfilled

Table 5.13.: Fulfillment of Evaluation Criteria in Category Testing (IV.1)

5.3.4.2. Documentation (Evaluation Category IV.2)

Automated documentation for *Spooq* is provided via a Python library called *Sphinx*. Relevant information is written as docstrings within *Spooq's* source code (EC IV.2.2). This convention makes interactive consultation possible via IDEs (integrated development environments), Python shells, and notebooks like *JupyterLab*. Executing **`make html`** generates or updates a web page which can be easily hosted to deliver documentation via a web browser (EC IV.2.1). Static PDF files can be created by **`make latexpdf`**, which utilizes the word processing engine of LaTeX (EC IV.2.1). The PDF documentation of *Spooq* is attached in the appendix at Section "Appendix A: Spooq Documentation." Two references have to be updated to include an ETL component into the documentation, whereas all relevant information is provided within the source code in the form of docstrings. Please refer to Section 5.2.4 for detailed examples.

Spooq offers its extensive documentation in various, accessible ways. Including new modules and classes to the documentation needs only minor adaptations. All evaluation criteria regarding the documentation of *Spooq* are fulfilled, as summarized in Table 5.14.

EC	Name	Status
IV.2.1	Support for different formats	Fulfilled
IV.2.2	Documentation by source code	Fulfilled

Table 5.14.: Fulfillment of Evaluation Criteria in Category Documentation (IV.2)

5.3.5. Summary

Spooq was checked for its functionality and efficacy of its big data processing capabilities. It provides extractors for different source formats, multiple transformers, and a loader class for Hive databases. Scalability abilities were found suitable, as *Spooq* utilizes Apache Spark in the background, which is built with parallel processing and horizontal scaling in mind. Switching the pipeline design focus from writing software to defining business logic decreases the complexity. *Spooq* relies on extended parameterization to achieve this goal. Its support for semi-automatic configuration by reasoning can decrease the complexity even further if an external expert system and metadata repository is utilized. The engineering principle of broad applicability was illustrated by employing a *Spooq* pipeline in a Hadoop environment and a cloud-based Spark cluster. Evolvability, as another principle of software engineering, was measured by the necessary effort to implement new ETL components that are fully integrated, tested, and documented without violating any dependencies to existing components. The quality of *Spooq's* data pipelines was asserted by measuring its code coverage by unit tests and documentation. Table 5.15 gives an overview over all evaluation criteria with their respective status of fulfillment.

The next and last part of this thesis will discuss its results and conclude its effects, limits, and prospects.

EC			Name	Status
I			**Providing ETL Functionality for Big Data**	
I	1		Functionality	
I	1	1	Implementation of one data extractor	Fulfilled
I	1	2	Implementation of one data transformer	Fulfilled
I	1	3	Implementation of one data loader	Fulfilled
I	1	4	Implementation of one pipeline object	Fulfilled
I	2		Scalabiltity	
I	2	1	Support for parallel computing	Fulfilled
I	2	2	Support for horizontal scaling	Fulfilled
I	2	3	Compatibility with cloud services	Fulfilled
II			**Decrease Complexity of Data Pipelines**	
II	1		Parameterizable	
II	1	1	Configure daily batch-processing pipeline via parameters	Fulfilled
II	1	2	Configure ad hoc data preparation pipeline via parameters	Fulfilled
II	2		Semi-Automatic Configuration by Reasoning	
II	2	1	Configure daily batch-processing pipeline via reasoning	Partly fulfilled
II	2	2	Configure ad hoc data preparation pipeline via reasoning	Partly fulfilled
III			**Conform with Standards and Best Practices**	
III	1		Code-Focus	
III	1	1	Provide code-based interface	Fulfilled
III	2		Broad Applicability	
III	2	1	Compatibility with stand-alone Spark environment	Fulfilled
III	2	2	Compatibility with on-premises Cloudera Hadoop cluster	Fulfilled
III	2	3	Compatibility with cloud-based Databricks cluster	Fulfilled
III	3		Evolvability	
III	3	1	Add additional data extractor	Fulfilled
III	3	2	Add additional data transformer	Fulfilled
III	3	3	Add additional data loader	Fulfilled
IV			**Increase Quality of Data Pipelines**	
IV	1		Testing	
IV	1	1	At least 75 percent code-coverage	Fulfilled
IV	1	2	Effort to write tests for ETL components	Fulfilled
IV	2		Documentation	
IV	2	1	Support for different formats	Fulfilled
IV	2	2	Documentation by source code	Fulfilled

Table 5.15.: Fulfillment of Evaluation Criteria

Part III.

Discussion and Conclusion

6. Discussion

This section examines specific aspects of the software research and development project and its resulting IT artifact. It starts with the communication of the research's results to different groups of interest. The primary medium of communication, open-sourcing the software library, is also explained in this section. *Spooq's* evaluation results are discussed afterward. The established objectives are discussed and compared against the outcome of this thesis. Ideas for the next design cycle are described in the last part of this section.

6.1. Communication

Communicating the results of this academic project represents the last activity according to the design science research methodology introduced by Peffers et al. (2007).

The first recipients of information about the refined version of *Spooq* have been the author's colleagues. New employees who joined the Data Science and Data Engineering team at Runtastic in the past few months were given a short overview of what issues we faced concerning the design of big data pipelines in our Hadoop cluster. *Spooq* was introduced to them as a software library, which makes writing ETL processes simpler and keeps conformity among all pipelines. Documentation and tests were emphasized as additional benefits of *Spooq* pipelines compared to basic scripts. The new colleagues of the author were appreciative of the advanced development phase and

the wholesome approach. For more details, they were given a link to *Spooq's* documentation and its source code.

The management at Runtastic was notified that the development of *Spooq* with compatibility for Spark 2 had reached beta status. A future project of Runtastic is to move its data processing and analytics backend into the cloud. Databricks workspaces will be used with Microsoft Azure web services as a basis. The head of data engineering plans to utilize *Spooq* for future cloud-based data pipelines.

Long time colleagues and external consultants got to know about *Spooq* through the evolutionary development processes. They were given access to the most up-to-date source code and *Spooq's* documentation.

The primary channel of communicating about *Spooq* is to open-source it. A GitHub repository has been created at https://github.com/breaka84/spooq. *Spooq's* current code state, which is subject to this thesis, is preserved in the git branch *state_for_thesis*. The master branch of the GitHub repository will act as the central place for future development, commenting, and releasing of *Spooq*. HTML documentation will be provided via the *Read The Docs* website at https://spooq.readthedocs.io.

6.2. Interpretation of Spooq's Evaluation

The criteria for evaluating *Spooq* have been inferred from the problem context and best practices of data and software engineering. Four main aspects were defined and subsequently evaluated based on demonstrative examples.

Providing ETL Functionality for Big Data (Evaluation Category I)

Spooq's implemented functionality features three extractor classes, five transformer classes, and one loader class. The evaluation

criteria previously defined require one class per ETL step. Scalability to cope with big data is implicitly achieved as *Spooq* uses the computation engine of big data-focused Apache Spark.

The produced artifact fully met the functionality and scalability criteria.

Decrease Complexity of Data Pipelines (Evaluation Category II)

Abstracting away processing details reduces the complexity of data pipelines. There is no need to know any internals of Spark to use *Spooq*. All functionality can be expressed and configured via parameterization of its ETL components. The complexity can be lowered significantly by utilizing an expert system and semi-automate the construction of data pipelines. With this setup, all relevant data engineering expertise is outsourced to a knowledge base, and providing a few attributes about the use case is sufficient to extract, transform, and load data.

The capability to parameterize *Spooq's* ETL components is fully implemented. Automating the design of data pipelines is supported via an interface, but the logic itself is not part of *Spooq*. Strictly speaking, *Spooq*, representing the resulting IT artifact, does not have the internal capabilities for semi-automatic configuration. However, a capable expert system called *spooq_rules* was implemented to prove the concept and to serve as a reference. *Spooq*, in combination with *spooq_rules* as its by-product, allows for semi-automatic configuration and process automation.

The critera for parameterization support are fully fulfilled. Semi-automatic configuration by reasoning is only possible with an additional component, which is not part of *Spooq*.

Conform with Standards and Best Practices (Evaluation Category III)

Choosing Python as the interface language and utilizing Scala

for the computation confirms with data-oriented standards and best practices. *Spooq* focuses on code rather than on graphical interfaces. This focal point facilitates its portability and allows the software library to run on single computers, up-to-date Hadoop-based distributions, and cloud-based Spark environments, without adaptions to its source code. Implementing additional extractor, transformer, or loader components is possible without any undesired side-effects.

The two criteria concerning the broad applicability of *Spooq's* are fully achieved by utilizing Python and Spark. The evaluation criterion about evolvability is also met.

Increase Quality of Data Pipelines (Evaluation Category IV)

The codebase of *Spooq*, including all implemented ETL classes, is unit tested with an average coverage of 90 percent, which increases the reliability and demonstrates the behavior of *Spooq*. The documentation can be created and updated automatically from the source code and is provided via a website, a PDF file, and via dynamic code inspection.

Both criteria, concerning testing and documentation, are completely satisfied.

Three out of four evaluation criteria categories were fully met. The evaluation criteria concerning semi-automatic configuration by reasoning (EC II.2.1 and EC II.2.2) were only partly met. The resulting IT artifact provides support in the form of an interface but does not include the necessary inference components. However, a prototype to demonstrate the potential was implemented.

In total, 20 out of 22 evaluation criteria were fully met. The remaining two criteria can be interpreted as partially satisfied, as the requirements were fulfilled with the help of an external system, which

was developed in parallel to this thesis' IT artifact, but not directly included.

6.3. Achievement of Research Objectives

The primary problem that this thesis addresses is the high complexity of data transformations in data lakes. Proven standards and best practices are scarce, which can result in an increased effort to design data pipelines with lower effective quality. Consequently, business opportunities are missed because of delays in data preparation. The following items examine whether the problem could be solved or mitigated through *Spooq*, the developed IT artifact.

Increased complexity due to the unlimited variety of data
: The variety of data content and structure lies unchanged. Complexity due to the diversity of data types is neither addressed nor solved by *Spooq*. This situation primarily concerns the definition of business logic which is still necessary for utilizing *Spooq*, either in explicit form via manually designing pipelines, or implicitly, if an external expert system is used.

Increased complexity due to the software stack
: *Spooq* utilizes Apache Spark, which is a conventional computation engine for big data processing. Abstracting away the internal processing methods and functions of Spark shifts the focus of designing data pipelines from writing software to defining business logic. Data engineers and data scientists can utilize *Spooq* to extract raw data, transform it into a suitable format, and store or receive the result. Only business logic has to be of concern for these operators, as *Spooq* is fully parameterizable.

No established standards to provide conformity
: Apache Spark, on which *Spooq* is based, provides extensive support for various data transformations and conversions. It covers

many functionalities that are necessary for ETL processes. On-premises data-centric environments and cloud providers generally feature a Spark cluster, which allows the utilization of *Spooq*. This fact enables the software library to provide conformity over different environments and, therefore, companies and institutions as well. Within a company, *Spooq* facilitates conformity as it serves multiple use cases. Through its parameterization concept, little code has to be implemented, and pipelines look similar, except for the applied business logic.

Low quality of data pipelines

The reusable software component approach of *Spooq* allows to test all relevant parts of its code extensively. Its pre-selected set of functionalities leads to a limited scope of use cases, which is consequently less effort to cover by unit tests. *Spooq's* existing test suite can be easily extended with new tests, either if a new component is added or if present classes are adapted, which decreases the development effort for testing. The general structure of *Spooq's* classes uses an existing standard for documentation that is well-known among Python developers. The automated generation of documentation in the form of a website or PDF file lowers the effort to document ETL process steps.

The decreased effort for testing and documenting reduces the development time of data pipelines. It allows engineers to focus more on the quality of tests, documentation, and especially the pipelines to implement.

Missed business value due to long development times

The utilization of *Spooq* can shorten the time to develop ETL pipelines. Datasets are made available in a shorter period after their first storage in the data lake. Data scientists and business analysts are not dependent on software engineering specialists to extract data with additional attributes, which are missing in the general reporting backend. Business decisions can be taken earlier and with a lower chance of quality issues. Eventu-

ally, these benefits can influence the evolution and success of a company.

Spooq addresses the problem scopes that cause delayed data presentation. Through its ability to reduce complexity for building data pipelines, it shortens the development time and increases the quality of ETL processes by unit tests and extensive documentation. The primary research objectives of mitigating the problems, as mentioned earlier, are therefore met.

6.4. Next Design Cycle

The primary drivers for the evolution of *Spooq* are changes to its requirements and objectives. The deprecation of Python 2 requires support for Python 3 by *Spooq*. One possible option for the next design cycle is, therefore, to rebase the software library on Python 3. To keep backward compatibility, workarounds to maintain Python 2 support are necessary. The extended language compatibility will ensure the compatibility of *Spooq* with future versions of Spark in general and Spark environments in particular.

Another possibility for the next design cycle is to focus on further decreasing the complexity of building data pipelines. Utilizing an expert system to outsource the business logic to an external service decreases the complexity at application time immensely. *spooq_rules*, the application developed to demonstrate *Spooq's* support for semi-automatic configuration by reasoning, is currently only a proof-of-context. By extending and generalizing its functionality, it can become part of the *Spooq* library as a reference implementation, which is also ready to be utilized. This addition will increase the functionality of *Spooq* and shorten the development time of data pipelines even further.

7. Conclusion

The last section of this thesis recapitulates the software research and development project at hand. It begins with a summary of the research problem and continues with an introspection of its limitations, both of the research approach and its resulting artifact. The second last section outlines the stakeholders who can benefit from this study. Finally, ideas about future research topics, which can be based on the results of this project, are given.

7.1. Research Summary

Research Problem

Processing data within data lakes is a complex and complicated operation. Due to its late binding on schemata, converting raw data into table-like structures poses an effort every time the source data is accessed. Data scientists and business analysts need, therefore, often support from data engineers for atypical data requests. Data lakes generally consist of a series of open-source software components, which makes standardization difficult. Metadata support for data lineage and governance is, in many cases, only provided by proprietary products, which entails vendor lock-in. As a result, designing and implementing data pipelines takes a lot of effort and time. Due to the time constraints, testing and documentation often stay behind. Consequently, businesses have to delay their data-driven decisions and work with an error-prone data basis. Data-centric product

features are falling behind schedule or do not live up to their full potential.

The solution proposition, taken from the introduction section, reads as follows:

> The usage of this software library improves data pipeline development by utilizing ready-made code such that quality improves and implementation effort decreases in order to be able to generate more value in a shorter amount of time.

Methodology

The incremental software development method *evolutionary prototyping* was applied to produce the proposed IT artifact, called *Spooq*. The iterative implementation phases were assisted and augmented by activities taken from the compatible and suitable DSRM (design science research methodology) by Peffers et al. (2007).

Course of Action

The development of *Spooq* was initiated by trying to solve a practical problem (problem-centered initiation). The author has been experiencing the high complexity of data pipelines in data lakes through his work as a data engineer at Runtastic GmbH. As a result of conversations with his colleagues and own research, he compiled objectives that mitigate the problems and benefit affected stakeholders. Relevant evaluation criteria were derived from the principles of data and software engineering in general, and the problem-specific objectives in particular.

The design and development of the produced IT artifact were done in cyclic stages. The applied software engineering method is based on incremental development, which reuses the output of previous stages. After each iteration, the software library was demonstrated via code reviews, discussions, and productive application. The evaluation of insights gained from the demonstration phase effected revisions of earlier stages. Bugs and

errors called for starting another design and implementation iteration, while missing functionalities and changed requirements led to updating the objectives and re-entering the design and development step.

Research Result

The current version of *Spooq*, which is subject to this thesis, is 2.0. It features support for Spark 2, compatibility for semi-automatic configuration by reasoning, and general code refactoring. The software library assists data engineers, data scientists, and business analysts with extracting raw data, transforming it according to their needs, and return or persist resulting datasets. The produced artifact was fully open-sourced and can be found in its current state under https://github.com/breaka84/spooq/tree/state_for_thesis. Its respective documentation is hosted at https://spooq.readthedocs.io/en/state_for_thesis/.

7.2. Limitations

The author firmly believes that the results of this thesis can benefit multiple companies and institutions. However, there are limitations concerning the research process and the IT artifact, which are addressed in this section.

No Practical Application by Other Companies

The developed software library is currently used exclusively by the company Runtastic. One of the objectives of this thesis is to design an artifact that is applicable and beneficial to other companies and institutions as well. The evaluation of *Spooq's* broad applicability is solely based on inductive reasoning, which derives *Spooq's* usefulness to other companies by testing the technical feasibility in different software environments. Empirical case studies, with multiple companies, have to be performed to prove this hypothesis.

No Reasoning Engine Included
Inferring all necessary components and their parameters can decrease the complexity of constructing data pipelines significantly. *Spooq* provides an interface for the automation of pipelines via its `PipelineFactory` class. However, this feature depends on an external service, which is responsible for the structure and definitions of a data pipeline. *Spooq* does not include the means to reason over its parameters by itself.

No Support For Multiple Data Frames per Pipeline
One limiting factor of the current design of *Spooq's* pipelines is the lack of support for multiple data frames. A Pipeline object can take at maximum one extractor instance which delivers, per definition, a single dataset. All transformer classes share the strict parameter-list of input and output values, which limits them to single data objects. Loaders can also take only a single input dataset to persist it.

No Support For Dimensional Data Warehousing
Spooq is designed explicitly for ETL processes in data lakes with non-normalized datasets. However, utilizing data lakes as the primary storage for data does not prevent or exclude the usage of data warehouses. Data warehouses commonly use dimensional modeling and de-normalization to store and represent their data. *Spooq* does not know about *facts* or *dimensions*, nor does it support merges, look-ups, or joins to support dimensional modeling. Although, those transformations can be done manually by dropping back to pure Spark functions.

7.3. Potential Beneficiaries

There are two groups of people who profit from the IT artifact, which was developed for this software research and development project. The primary beneficiaries are within Runtastic, the author's work-

place. The second group covers other companies and institutions which already utilize Apache Spark within data lakes.

Runtastic

Data practitioners and colleagues of the author at Runtastic are already introduced to *Spooq*. Data engineers use it to design ETL pipelines. Data scientists load and transform raw data for analyses and exploration with the help of *Spooq*. The management at the company knows about the functionality and advantages of *Spooq* and plans its utilization for current and future projects. In general, Runtastic can make business decisions based on more recent data due to the decreased ETL process implementation effort. The data basis for reporting is less error-prone than using one-time scripts for data acquisition. Data scientists have faster access to extensive information, hidden in the raw input data.

Other Companies Employing Apache Spark Clusters

The challenges with big data processing in data lakes are not unique to Runtastic. Several companies have similar data, use cases, and requirements which allows their data engineers and data scientists to utilize *Spooq* within their environment directly. For businesses which have differing applications, *Spooq* can be adapted for their demands by adding necessary ETL components by their engineers. The primary contribution to the data engineering community is the open-sourcing of *Spooq's* code. It can be used as a guideline on how to write tested and documented PySpark libraries. Other developers may cherry-pick architectural ideas they deem useful and implement their own, *Spooq*-inspired ETL libraries.

7.4. Future Work

Topics for potential future research can be derived from the current limitations of this software research and development project. The

author finds the following subjects engaging and suitable for further studies based on the outcome of this thesis.

Deployment at Other Companies
The limited evaluation of *Spooq's* broad applicability does neither provide details about the efficacy nor about possible revisions necessary to the software's structure. Case studies about practical usage of companies are an option to evaluate this big data ETL library empirically.

Inclusion of Reasoning Component
The reasoning engine *spooq_rules* was implemented as a proof-of-concept for *Spooq's* ability to automate pipeline construction, if given appropriate information. An interesting approach is to augment *Spooq* itself with inference capabilities by including the reasoning service into the library.

Adaptation For Dimensional Modeling
Spooq's current functionality is strictly focused on typical ETL and ELT big data workloads, which does not take complex dependencies to other datasets into account. Enriching *Spooq* with capabilities for dimensional modeling would allow high compatibility with relational data warehouses. These abilities would increase the range of potential beneficiaries significantly.

Adding a Graphical User Interface
Although *Spooq* is designed with code interfaces in mind, adding a GUI (graphical user interface) can lower the entry hurdle for some data practitioners. This additional way of interaction would increase the range of interested communities and persons.

Bibliography

Adidas GmbH. (2016, March 3). *adidas Group Geschäftsbericht 2015*. Retrieved January 16, 2020, from https://www.adidas-group.com/media/filer_public/28/df/28df5eae-389a-4932-a8da-6ba2ef7a6922/2015_gb_de.pdf. (Cit. on p. 36)

Alter, S. (2006). Work Systems and IT Artifacts - Does the Definition Matter? *Communications of the Association for Information Systems*, *17*. https://doi.org/10.17705/1cais.01714 (cit. on p. 23)

Anderson, J. (2016a, February 4). *Is my developer team ready for big data? - O'Reilly Media*. Retrieved January 16, 2020, from https://www.oreilly.com/ideas/is-my-developer-team-ready-for-big-data. (Cit. on p. 37)

Anderson, J. (2016b, August 25). *On complexity in big data – O'Reilly*. Retrieved January 16, 2020, from https://www.oreilly.com/radar/on-complexity-in-big-data/. (Cit. on p. 37)

Anderson, J. (2018, June 27). *The Two Types of Data Engineering* (J. Anderson, Ed.). http://www.jesse-anderson.com/2018/06/the-two-types-of-data-engineering/. (Cit. on pp. 37, 55)

Anderson, J. (2019, April 9). *Why a data scientist is not a data engineer*. Retrieved December 5, 2019, from https://www.oreilly.com/ideas/why-a-data-scientist-is-not-a-data-engineer. (Cit. on p. 6)

Apache Software Foundation. (2018a). *Apache Mesos*. http://mesos.apache.org/. (Cit. on p. 67)

Apache Software Foundation. (2018b). *Apache Spark - Lightning-Fast Cluster Computing*. https://spark.apache.org/. (Cit. on p. 57)

Apache Software Foundation. (2018c). *Running Spark on Kubernetes*. https://spark.apache.org/docs/latest/running-on-kubernetes.html. (Cit. on p. 66)

Apache Software Foundation. (2018d). *Running Spark on YARN*. https://spark.apache.org/docs/latest/running-on-yarn.html. (Cit. on pp. 68, 74)

Apache Software Foundation. (2020, January 10). *Releases · apache/spark · GitHub*. Retrieved January 10, 2020, from https://github.com/apache/spark/releases. (Cit. on p. 29)

Applegate, L. M. (1999). Rigor and Relevance in MIS Research-Introduction. *MIS Quarterly*, *23*(1), 1–2. http://www.jstor.org/stable/249402 (cit. on p. 17)

Armbrust, M., Xin, R. S., Lian, C., Huai, Y., Liu, D., Bradley, J. K., Meng, X., Kaftan, T., Franklin, M. J., Ghodsi, A., & et al. (2015). Spark SQL: Relational Data Processing in Spark. *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, 1383–1394. https://doi.org/10.1145/2723372.2742797 (cit. on pp. 78, 79)

Baesens, B. (2014, July). *Analytics in a Big Data World: The Essential Guide to Data Science and Its Applications*. Wiley. (Cit. on p. 3).

Beck, K. M., Beedle, M., van Bennekum, A., Cockburn, A., Cunningham, W., Fowler, M., Grenning, J., Highsmith, J., Hunt, A., Jeffries, R., Kern, J., Marick, B., Martin, R. C., Mellor, S. J., Schwaber, K., Sutherland, J., & Thomas, D. (2013). Manifesto for Agile Software Development (cit. on p. 10).

Bischofberger, W., & Pomberger, G. (1992). *Prototyping-Oriented Softare Development - Concepts and Tools*. Springer Verlag. (Cit. on pp. 12, 13).

Brandl, G. (2019, October 15). *Overview — Sphinx 1.8.6+/6ef08a42d documentation*. Retrieved March 6, 2020, from https://www.sphinx-doc.org/en/1.8/. (Cit. on p. 121)

Buchanan, B. G., Davis, R., & Feigenbaum, E. A. (2006). Expert Systems: A Perspective from Computer Science. In K. A. Ericsson, N. Charness, P. J. Feltovich, & R. R. Hoffman (Eds.), *The Cambridge Handbook of Expertise and Expert Performance* (pp. 87–104). Cambridge University Press. https://doi.org/10.1017/CBO9780511816796.006. (Cit. on p. 84)

CERN. (2019, December 4). *Processing: What to record? | CERN*. Retrieved December 4, 2019, from https://home.cern/science/computing/processing-what-record. (Cit. on p. 3)

Chambers, B., & Zaharia, M. (2018, February 1). *Spark: The Definitive Guide: Big Data Processing Made Simple*. O'Reilly Media. (Cit. on pp. 63, 67, 68, 73, 74, 76–81, 164).

Chaudhuri, S., & Dayal, U. (1997). An Overview of Data Warehousing and OLAP Technology. *SIGMOD Rec.*, *26*(1), 65–74. https://doi.org/10.1145/248603.248616 (cit. on p. 4)

Chen, H. (2011). Editorial: Design science, grand challenges, and societal impacts. *ACM Transactions on Management Information Systems*, *2*(1). https://doi.org/10.1145/1929916.1929917 (cit. on p. 15)

Cloudera, Inc. (2015, November 1). *CDH 5.5.x Packaging and Tarball Information | 5.x | Cloudera Documentation*. Retrieved January 16, 2020, from https://docs.cloudera.com/documentation/enterprise/release-notes/topics/cdh_vd_cdh_package_tarball_55.html#topic_3. (Cit. on pp. 36, 37)

Crawford, C., Montoya, A., O'Connell, M., & Mooney, P. (2018, November 3). *2018 Kaggle ML & DS Survey | Kaggle*. Retrieved February 19, 2020, from https://www.kaggle.com/kaggle/kaggle-survey-2018. (Cit. on p. 75)

Cross, N. (1993). A History of Design Methodology. In M. J. de Vries, N. Cross, & D. P. Grant (Eds.), *Design Methodology and Relationships with Science* (pp. 15–27). Springer Netherlands. https://doi.org/10.1007/978-94-015-8220-9_2. (Cit. on p. 14)

Databricks Inc. (2020, March 21). *Databricks - Unified Data Analytics*. Retrieved March 21, 2020, from https://databricks.com/. (Cit. on p. 150)

Datenschutzbehörde, Ö. (2018). *Home - Austrian Data Protection Authority*. https://www.data-protection-authority.gv.at/. (Cit. on p. 54)

DM Review and SourceMedia. (2019, November 13). *Glossary*. DM Review and SourceMedia. Retrieved December 20, 2019, from http://www.dmreview.com/resources/glossary_keywordId_M.html. (Cit. on p. 18)

Douven, I. (2017, April 28). *Abduction (Stanford Encyclopedia of Philosophy)*. Retrieved February 25, 2020, from https://plato.stanford.edu/entries/abduction/#AbdGenIde. (Cit. on pp. 87, 88)

Drabas, T., & Lee, D. (2017). *Learning PySpark*. Packt Publishing. (Cit. on pp. 76, 77, 79, 80).

Droettboom, M. (2019, October 22). *Understanding JSON Schema — Understanding JSON Schema 7.0 documentation*. Retrieved February 6, 2020, from https://json-schema.org/understanding-json-schema/. (Cit. on p. 132)

Fang, H. (2015). Managing data lakes in big data era: What's a data lake and why has it became popular in data management ecosystem. *2015 IEEE International Conference on Cyber Technology in Automation, Control, and Intelligent Systems (CYBER)*, 820–824. https://doi.org/10.1109/CYBER.2015.7288049 (cit. on pp. 4–6)

Feigenbaum, E. A. (1981). Expert systems in the 1980s. *State of the art report on machine intelligence. Maidenhead: Pergamon-Infotech* (cit. on p. 82).

Forgy, C. L. (1979). *On the efficient implementation of production systems* (Doctoral dissertation). Carnegie-Mellon University. (Cit. on pp. 92, 93, 97).

Gauch Jr, H. G. (2002). *Scientific Method in Practice*. Cambridge University Press. https://doi.org/10.1017/CBO9780511815034. (Cit. on p. 9)

Giarratano, J., & Riley, G. (2005). *Expert systems : principles and programming* (Fourth). Thomson Course Technology. (Cit. on pp. 82–85, 87, 89, 90, 92).

Giarratano, J. C. (2015, July 1). *CLIPS Online Documentation*. Retrieved February 21, 2020, from http://clipsrules.sourceforge.net/OnlineDocs.html. (Cit. on pp. 91, 93, 94)

Goldman, T. (2017). *Data Fingerprinting – The Magic is Finally Revealed*. https://www.waterlinedata.com/blog/data-fingerprinting-the-magic-is-finally-revealed/. (Cit. on p. 53)

Golfarelli, M., & Rizzi, S. (2009). *Data Warehouse Design: Modern Principles and Methodologies*. McGraw-Hill, Inc. (Cit. on pp. 52–54).

Gregor, S., & Hevner, A. R. (2013). Positioning and Presenting Design Science Research for Maximum Impact. *MIS Q.*, *37*(2), 337–356. https://doi.org/10.25300/MISQ/2013/37.2.01 (cit. on pp. 22–24)

Hajmoosaei, A., Kashfi, M., & Kailasam, P. (2011). Comparison plan for data warehouse system architectures. *The 3rd International Conference on Data Mining and Intelligent Information Technology Applications*, 290–293 (cit. on p. 4).

Hevner, A. R., March, S. T., Park, J., & Ram, S. (2004). Design Science in Information Systems Research. *MIS Q.*, *28*(1), 75–105. http://dl.acm.org/citation.cfm?id=2017212.2017217 (cit. on pp. 10, 14–18, 20, 22, 23)

Hindman, B., Konwinski, A., Zaharia, M., Ghodsi, A., Joseph, A. D., Katz, R., Shenker, S., & Stoica, I. (2011). Mesos: A Platform for Fine-grained Resource Sharing in the Data Center. *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, 295–308. http://dl.acm.org/citation.cfm?id=1972457.1972488 (cit. on p. 67)

Jing Han, Haihong E, Guan Le, & Jian Du. (2011). Survey on NoSQL database. *2011 6th International Conference on Pervasive Computing and Applications*, 363–366. https://doi.org/10.1109/ICPCA.2011.6106531 (cit. on p. 5)

Karau, H., & Warren, R. (2017). *High Performance Spark: Best Practices for Scaling and Optimizing Apache Spark*. O'Reilly Media. (Cit. on pp. 75, 77–79, 164).

Katz, Y., Gebhardt, D., & Sullice, G. (2020, February 26). *JSON:API — A specification for building APIs in JSON*. Retrieved March 3, 2020, from https://jsonapi.org/format/. (Cit. on p. 110)

Kimball, R., & Caserta, J. (2004). The Data Warehouse ETL Toolkit: Practical Techniques for Extracting, Cleaning, Conforming, and Delivering Data (cit. on pp. 6, 7, 36).

Kimball, R., & Ross, M. (2013). *The Data Warehouse Toolkit: The Definitive Guide to Dimensional Modeling*. John Wiley & Sons. (Cit. on pp. 36, 52, 54–56).

Kleppmann, M. (2017). *Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems*. O'Reilly Media. (Cit. on pp. 43, 44, 132).

Lee, A. (1999). Inaugural Editor's Comments. *MIS Quarterly*, *23*(1), v–xi. http://www.jstor.org/stable/249400 (cit. on p. 17)

Lee, D., & Damji, J. (2016, June 22). *Apache Spark Key Terms, Explained*. https://databricks.com/blog/2016/06/22/apache-spark-key-terms-explained.html. (Cit. on pp. 79, 80)

Loeliger, J., & McCullough, M. (2012). *Version Control with Git: Powerful tools and techniques for collaborative software development*. O'Reilly Media. (Cit. on p. 57).

March, S. T., & Smith, G. F. (1995). Design and natural science research on information technology. *Decision support systems*, *15*(4), 251–266 (cit. on p. 22).

McRitchie, V. (2018, September 17). *Project 1 - Classifying Iris Flowers*. Retrieved February 24, 2020, from http://rstudio-pubs-static.s3.amazonaws.com/420656_c17c8444d32548eba6f894bcbdffcaab.html. (Cit. on p. 86)

Moseley, B., & Marks, P. (2006). Out of the Tar Pit. *SOFTWARE PRACTICE ADVANCEMENT (SPA)* (cit. on p. 44).

Nandi, A. (2015). *Spark for Python Developers*. Packt Publishing. (Cit. on pp. 76, 78, 79).

Nyberg, C., & Shah, M. (2018). *Sort Benchmark Home Page*. http://sortbenchmark.org/. (Cit. on p. 64)

Offermann, P., Blom, S., Schönherr, M., & Bub, U. (2010). Artifact Types in Information Systems Design Science – A Literature Review. In R. Winter, J. L. Zhao, & S. Aier (Eds.), *Global Perspectives on Design Science Research* (pp. 77–92). Springer Berlin Heidelberg. (Cit. on p. 23).

Oliveira, B. (2018, November 11). *GitHub - pytest-dev/pytest: The pytest framework makes it easy to write small tests, yet scales to support complex functional testing*. Retrieved March 5, 2020, from https://docs.pytest.org/en/3.10.1/. (Cit. on p. 118)

Österle, H., Winter, R., & Brenner, W. (2010). *Gestaltungsorientierte Wirtschaftsinformatik: ein Plädoyer für Rigor und Relevanz*. Infowerk. (Cit. on pp. 10, 15).

Pasupuleti, P., & Purra, B. (2015). *Data Lake Development with Big Data*. Packt Publishing. (Cit. on pp. 4–6, 38).

Peffers, K., Tuunanen, T., Rothenberger, M. A., & Chatterjee, S. (2007). A Design Science Research Methodology for Information Systems Research. *Journal of Management Information Systems*, *24*, 45–78 (cit. on pp. 7, 9, 18–21, 24, 25, 28, 177, 186).

Pérez, R. A. M. (2019, November 16). *Experta Documentation*. Retrieved February 21, 2020, from https://experta.readthedocs.io/en/stable/. (Cit. on pp. xv, 93–97)

Poggi, N. (2017, October 31). *The State of Spark in the Cloud with Nicolas Poggi*. Retrieved March 21, 2020, from https://www.slideshare.net/SparkSummit/the-state-of-spark-in-the-cloud-with-nicolas-poggi. (Cit. on p. 168)

Pomberger, G., Bischofberger, W. R., Kolb, D., Pree, W., & Schlemm, H. (1991). Prototyping-Oriented Software Development - Concepts and Tools. *Structured Programming*, *12*, 43–60 (cit. on pp. 11, 12).

Python Software Foundation. (2020). *Sunsetting Python 2 | Python.org*. Retrieved January 10, 2020, from https://www.python.org/doc/sunset-python-2/. (Cit. on p. 29)

Ravat, F., & Zhao, Y. (2019). Data Lakes: Trends and Perspectives. In S. Hartmann, J. Küng, S. Chakravarthy, G. Anderst-Kotsis, A. M. Tjoa, & I. Khalil (Eds.), *Database and Expert Systems Applications* (pp. 304–313). Springer International Publishing. (Cit. on pp. 5–7).

Reinsel, D., Gantz, J., & Rydning, J. (2018). Data age 2025: the digitization of the world from edge to core. https://www.seagate.com/files/www-content/our-story/trends/files/idc-seagate-dataage-whitepaper.pdf (cit. on p. 3)

Reitz, K. (2018, November 27). *Pipenv: Python Dev Workflow for Humans — pipenv 2018.11.27.devo documentation*. Retrieved March 5, 2020, from https://pipenv.pypa.io/en/latest/. (Cit. on p. 98)

Ross, D. T., Goodenough, J. B., & Irvine, C. A. (1975). Software Engineering: Process, Principles, and Goals. *Computer*, *8*(5), 17–27. https://doi.org/10.1109/C-M.1975.218952 (cit. on pp. 22, 42–44)

runtastic GmbH. (2019a, May). *Facts & Figures | Runtastic Career*. Retrieved January 16, 2020, from https://www.runtastic.com/career/facts-about-runtastic/. (Cit. on p. 36)

runtastic GmbH. (2019b, June 27). *Run For Oceans 2019 • 12,6 Mio. Kilometer für den Schutz der Meere*. Retrieved January 16, 2020, from https://www.runtastic.com/blog/de/run-for-the-oceans-rueckblick/. (Cit. on p. 36)

Ryza, S., Laserson, U., Owen, S., & Wills, J. (2017). *Advanced Analytics with Spark: Patterns for Learning from Data at Scale*. O'Reilly Media. (Cit. on p. 164).

Sadalage, P., & Fowler, M. (2012). NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence (cit. on p. 5).

Sakr, S., Liu, A., Batista, D. M., & Alomari, M. (2011). A Survey of Large Scale Data Management Approaches in Cloud Environments. *IEEE Communications Surveys Tutorials*, *13*(3), 311–336. https://doi.org/10.1109/SURV.2011.032211.00087 (cit. on p. 5)

Santos, W. d., Carvalho, L. F. M., de P. Avelar, G., Silva, Á., Ponce, L. M., Guedes, D., & Meira, W. (2017). Lemonade: A Scalable and Efficient Spark-Based Platform for Data Analytics. *Proceedings of the 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, 745–748. https://doi.org/10.1109/CCGRID.2017.142 (cit. on pp. 37, 38)

Sasikumar, M., Ramani, S., Raman, S. M., Anjaneyulu, K., & Chandrasekar, R. (2007). A practical introduction to rule based expert systems. *New Delhi: Narosa Publishing House* (cit. on pp. 82–84, 86–89, 91, 92).

Semlinger, M., & Litzel, N. (2016, February 23). *Consol führt Data Lake bei Runtastic ein*. Retrieved January 16, 2020, from https://www.bigdata-insider.de/consol-fuehrt-data-lake-bei-runtastic-ein-a-522133/. (Cit. on p. 36)

Sharma, B. (2018). *Architecting data lakes : data management architectures for advanced business use cases*. O'Reilly Media. (Cit. on pp. 5, 6).

Shute, J., Oancea, M., Ellner, S., Handy, B., Rollins, E., Samwel, B., Vingralek, R., Whipkey, C., Chen, X., Jegerlehner, B., Littlefield, K., & Tong, P. (2012). F1 - The Fault-Tolerant Distributed RDBMS Supporting Google's Ad Business [Talk given at SIGMOD 2012]. *SIGMOD* (cit. on p. 4).

Simon, H. A. (1996). *The Sciences of the Artificial (3rd Ed.)* MIT Press. (Cit. on p. 17).

Stack Exchange, Inc. (2020a). *StackOverflow: Questions tagged [json]*. Retrieved March 30, 2020, from https://stackoverflow.com/questions/tagged/json. (Cit. on p. 132)

Stack Exchange, Inc. (2020b). *StackOverflow: Questions tagged [xml]*. Retrieved March 30, 2020, from https://stackoverflow.com/questions/tagged/xml. (Cit. on p. 132)

Stack Exchange, Inc. (2020c). *StackOverflow: Questions tagged [yaml]*. Retrieved March 30, 2020, from https://stackoverflow.com/questions/tagged/yaml. (Cit. on p. 132)

Szalay, A. S., & Blakeley, J. A. (2009). Gray's laws: database-centric computing in science. *The Fourth Paradigm* (cit. on p. 4).

Thomsen, C., & Bach Pedersen, T. (2009). Pygrametl: A Powerful Programming Framework for Extract-Transform-Load Programmers. *Proceedings of the ACM Twelfth International Workshop on Data Warehousing and OLAP*, 49–56. https://doi.org/10.1145/1651291.1651301 (cit. on p. 56)

Vaughan-Nichols, S. J. (2018, September 27). *Linux now dominates Azure | ZDNet*. Retrieved March 17, 2020, from https://www.zdnet.com/article/linux-now-dominates-azure/. (Cit. on p. 147)

Vavilapalli, V. K., Murthy, A. C., Douglas, C., Agarwal, S., Konar, M., Evans, R., Graves, T., Lowe, J., Shah, H., Seth, S., Saha, B., Curino, C., O'Malley, O., Radia, S., Reed, B., & Baldeschwieler, E. (2013). Apache Hadoop YARN: Yet Another Resource Negotiator. *Proceedings of the 4th Annual Symposium on Cloud Computing*, 5:1–5:16. https://doi.org/10.1145/2523616.2523633 (cit. on pp. 68, 69, 71–73)

Welch, R. (2018). *GDPR's Impact on BI (Part 1 in a Series)*. https://tdwi.org/articles/2018/06/04/biz-all-gdpr-impact-on-bi-1.aspx. (Cit. on p. 54)

White, T. (2015). *Hadoop: The Definitive Guide: Storage and Analysis at Internet Scale*. O'Reilly Media. (Cit. on pp. 69, 70).

Wilde, T., & Hess, T. (2007). Forschungsmethoden der Wirtschaftsinformatik. *WIRTSCHAFTSINFORMATIK*, *49*(4), 280–287. https://doi.org/10.1007/s11576-007-0064-z (cit. on pp. 10, 21)

Winter, R. (2008). Design science research in Europe. *European Journal of Information Systems*, *17*(5), 470–475. https://doi.org/10.1057/ejis.2008.44 (cit. on p. 10)

Wolfgang Bartel, G. S., Stefan Schwarz. (2013). Der ETL-Prozess des Data Warehousing. In R. Jung & R. Winter (Eds.), *Data Ware-*

housing Strategie: Erfahrungen, Methoden, Visionen (pp. 43–60). Springer-Verlag. (Cit. on p. 52).

Yelp Inc. (2020, March 13). *Yelp Dataset JSON*. Retrieved March 13, 2020, from https://www.yelp.com/dataset/documentation/main. (Cit. on pp. 135–138)

Zaharia, M. (2016). An Architecture for Fast and General Data Processing on Large Clusters. https://doi.org/10.1145/2886107.2886113 (cit. on pp. 66, 164)

Zaharia, M. (2020, March 2). *Matei Zaharia - Curriculum Vitæ*. Retrieved March 21, 2020, from https://cs.stanford.edu/people/matei/cv.pdf. (Cit. on p. 150)

Zaharia, M., Chowdhury, M., Franklin, M. J., Shenker, S., & Stoica, I. (2010). Spark: Cluster Computing with Working Sets. *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, 10–10. http://dl.acm.org/citation.cfm?id=1863103.1863113 (cit. on pp. 58–61, 64, 65, 164)

Zaharia, M., Xin, R. S., Wendell, P., Das, T., Armbrust, M., Dave, A., Meng, X., Rosen, J., Venkataraman, S., Franklin, M. J., Ghodsi, A., Gonzalez, J., Shenker, S., & Stoica, I. (2016). Apache Spark: A Unified Engine for Big Data Processing. *Commun. ACM*, *59*(11), 56–65. https://doi.org/10.1145/2934664 (cit. on pp. 58, 59, 61–65, 164)

Appendices

Appendix A: Spooq Documentation

Spooq2 Documentation

Release 2.0.0b0

David Hohensinn

Dec 27, 2020

Contents

Spooq is your PySpark based helper library for ETL data ingestion pipeline in Data Lakes.

Extractors, Transformers, and Loaders are independent components which can be plugged-in into a pipeline instance or used separately.

CHAPTER 1

Table of Content

1.1 Installation / Deployment

1.1.1 Build egg file

```
$ cd spooq2
$ python setup.py bdist_egg
```

The output is stored as *dist/Spooq2-<VERSION_NUMBER>-py2.7.egg*

1.1.2 Build zip file

```
$ cd spooq2
$ rm temp.zip
$ zip -r temp.zip src/spooq2
$ mv temp.zip Spooq2_$(grep "__version__" src/spooq2/_version.py | \
    cut -d " " -f 3 | tr -d \").zip
```

The output is stored as *Spooq2-<VERSION_NUMBER>.zip*.

1.1.3 Include pre-build package (egg or zip) with Spark

For Submitting or Launching Spark:

```
$ pyspark --py-files Spooq2-<VERSION_NUMBER>.egg
```

The library still has to be imported in the pyspark application!

Within Running Spark Session:

```
>>> sc.addFile("Spooq2-<VERSION_NUMBER>.egg")
>>> import spooq2
```

1.1.4 Install local repository as package

```
$ cd spooq2
$ python setup.py install
```

1.1.5 Install Spooq2 directly from git

```
$ pip install git+https://github.com/breaka84/spooq@master
```

1.1.6 Development, Testing, and Documenting

Please refer to *Setup for Development, Testing, Documenting*.

1.2 Examples

1.2.1 JSON Files to Partitioned Hive Table

Sample Input Data:

```
{
  "id": 18,
  "guid": "b12b59ba-5c78-4057-a998-469497005c1f",
  "attributes": {
    "first_name": "Jeannette",
    "last_name": "O'Loghlen",
    "gender": "F",
    "email": "gpirri3j@oracle.com",
    "ip_address": "64.19.237.154",
    "university": "",
    "birthday": "1972-05-16T22:17:41Z",
    "friends": [
      {
        "first_name": "Noémie",
        "last_name": "Tibbles",
        "id": 9952
      },
      {
        "first_name": "Bérangère",
        "last_name": null,
        "id": 3391
      },
      {
        "first_name": "Danièle",
        "last_name": null,
        "id": 9637
      },
      {
        "first_name": null,
        "last_name": null,
        "id": 9939
      },
      {
        "first_name": "Anaëlle",
        "last_name": null,
        "id": 18994
      }
    ]
  },
  "meta": {
```

(continues on next page)

(continued from previous page)

```
    "created_at_sec": 1547371284,
    "created_at_ms": 1547204429000,
    "version": 24
  }
}
```

Sample Output Tables

Table 1: Table **"user"**

id	guid	forename	surname	gender	has_email	created_at
18	"b12b59ba..."	"Jeannette"	"O"Loghlen"	"F"	"1"	1547204429
...	...	...	...	...	...	...

Table 2: Table **"friends_mapping"**

id	guid	friend_id	created_at
18	b12b59ba...	9952	1547204429
18	b12b59ba...	3391	1547204429
18	b12b59ba...	9637	1547204429
18	b12b59ba...	9939	1547204429
18	b12b59ba...	18994	1547204429
...	...	...	...

Application Code for Updating the Users Table

```
from spooq2.pipeline import Pipeline
import spooq2.extractor as E
import spooq2.transformer as T
import spooq2.loader as L

users_mapping = [
    ("id",              "id",                   "IntegerType"),
    ("guid",            "guid",                 "StringType"),
    ("forename",        "attributes.first_name", "StringType"),
    ("surename",        "attributes.last_name",  "StringType"),
    ("gender",          "attributes.gender",     "StringType"),
    ("has_email",       "attributes.email",      "StringBoolean"),
    ("created_at",      "meta.created_at_ms",    "timestamp_ms_to_s"),
]

users_pipeline = Pipeline()

users_pipeline.set_extractor(E.JSONExtractor(input_path="tests/data/schema_v1/
↪sequenceFiles"))

users_pipeline.add_transformers(
    [
        T.Mapper(mapping=users_mapping),
        T.ThresholdCleaner(
            range_definitions={"created_at": {"min": 0, "max": 1580737513, "default":␣
↪None}}
        ),
        T.NewestByGroup(group_by="id", order_by="created_at"),
    ]
)

users_pipeline.set_loader(
    L.HiveLoader(
        db_name="users_and_friends",
```

(continues on next page)

(continued from previous page)

```
        table_name="users",
        partition_definitions=[
            {"column_name": "dt", "column_type": "IntegerType", "default_value":␣
↪20200201}
        ],
        repartition_size=10,
    )
)

users_pipeline.execute()
```

Application Code for Updating the Friends_Mapping Table

```
from spooq2.pipeline import Pipeline
import spooq2.extractor as E
import spooq2.transformer as T
import spooq2.loader as L


friends_mapping = [
    ("id",          "id",                  "IntegerType"),
    ("guid",        "guid",                "StringType"),
    ("friend_id",   "friend.id",           "IntegerType"),
    ("created_at",  "meta.created_at_ms",  "timestamp_ms_to_s"),
]

friends_pipeline = Pipeline()

friends_pipeline.set_extractor(E.JSONExtractor(input_path="tests/data/schema_v1/
↪sequenceFiles"))

friends_pipeline.add_transformers(
    [
        T.NewestByGroup(group_by="id", order_by="meta.created_at_ms"),
        T.Exploder(path_to_array="attributes.friends", exploded_elem_name="friend"),
        T.Mapper(mapping=friends_mapping),
        T.ThresholdCleaner(
            range_definitions={"created_at": {"min": 0, "max": 1580737513, "default":␣
↪None}}
        ),
    ]
)

friends_pipeline.set_loader(
    L.HiveLoader(
        db_name="users_and_friends",
        table_name="friends_mapping",
        partition_definitions=[
            {"column_name": "dt", "column_type": "IntegerType", "default_value":␣
↪20200201}
        ],
        repartition_size=20,
    )
)

friends_pipeline.execute()
```

Application Code for Updating Both, the Users and Friends_Mapping Table, at once

This script extracts and transforms the common activities for both tables as they share the same input data set. Caching the dataframe avoids redundant processes and reloading when an action is executed (the load step f.e.). This could

have been written with pipeline objects as well (by providing the Pipeline an `input_df` and/or `output_df` to bypass extractors and loaders) but would have led to unnecessary verbosity. This example should also show the flexibility of Spooq2 for activities and steps which are not directly supported.

```
import spooq2.extractor as E
import spooq2.transformer as T
import spooq2.loader as L

mapping = [
    ("id",              "id",                   "IntegerType"),
    ("guid",            "guid",                 "StringType"),
    ("forename",        "attributes.first_name", "StringType"),
    ("surename",        "attributes.last_name",  "StringType"),
    ("gender",          "attributes.gender",     "StringType"),
    ("has_email",       "attributes.email",      "StringBoolean"),
    ("created_at",      "meta.created_at_ms",    "timestamp_ms_to_s"),
    ("friends",         "attributes.friends",    "as_is"),
]

"""Transformations used by both output tables"""
common_df = E.JSONExtractor(input_path="tests/data/schema_v1/sequenceFiles").extract()
common_df = T.Mapper(mapping=mapping).transform(common_df)
common_df = T.ThresholdCleaner(
    range_definitions={"created_at": {"min": 0, "max": 1580737513, "default": None}}
).transform(common_df)
common_df = T.NewestByGroup(group_by="id", order_by="created_at").transform(common_df)
common_df.cache()

"""Transformations for users_and_friends table"""
L.HiveLoader(
    db_name="users_and_friends",
    table_name="users",
    partition_definitions=[
        {"column_name": "dt", "column_type": "IntegerType", "default_value": 20200201}
    ],
    repartition_size=10,
).load(common_df.drop("friends"))

"""Transformations for friends_mapping table"""
friends_df = T.Exploder(path_to_array="friends", exploded_elem_name="friend").
↪transform(
    common_df
)
friends_df = T.Mapper(
    mapping=[
        ("id",          "id",          "IntegerType"),
        ("guid",        "guid",        "StringType"),
        ("friend_id",   "friend.id",   "IntegerType"),
        ("created_at",  "created_at",  "IntegerType"),
    ]
).transform(friends_df)
L.HiveLoader(
    db_name="users_and_friends",
    table_name="friends_mapping",
    partition_definitions=[
        {"column_name": "dt", "column_type": "IntegerType", "default_value": 20200201}
    ],
    repartition_size=20,
).load(friends_df)
```

1.3 Extractors

Extractors are used to fetch, extract and convert a source data set into a PySpark DataFrame. Exemplary extraction sources are **JSON Files** on file systems like HDFS, DBFS or EXT4 and relational database systems via **JDBC**.

1.3.1 JSON Files

class JSONExtractor(*input_path=None*, *base_path=None*, *partition=None*)

Bases: *spooq2.extractor.extractor.Extractor*

The JSONExtractor class provides an API to extract data stored as JSON format, deserializes it into a PySpark dataframe and returns it. Currently only single-line JSON files are supported, stored either as textFile or sequenceFile.

Examples

```
>>> from spooq2 import extractor as E
```

```
>>> extractor = E.JSONExtractor(input_path="tests/data/schema_v1/sequenceFiles")
>>> extractor.input_path == "tests/data/schema_v1/sequenceFiles" + "/*"
True
```

```
>>> extractor = E.JSONExtractor(
>>>     base_path="tests/data/schema_v1/sequenceFiles",
>>>     partition="20200201"
>>> )
>>> extractor.input_path == "tests/data/schema_v1/sequenceFiles" + "/20/02/01" +
↪"/*"
True
```

Parameters

- **input_path** (str) – The path from which the JSON files should be loaded ("/*" will be added if omitted)
- **base_path** (str) – Spooq tries to infer the input_path from the base_path and the partition if the input_path is missing.
- **partition** (str or int) – Spooq tries to infer the input_path from the base_path and the partition if the input_path is missing. Only daily partitions in the form of "YYYYMMDD" are supported. e.g., "20200201" => <base_path> + "/20/02/01/*"

Returns The extracted data set as a PySpark DataFrame

Return type pyspark.sql.DataFrame

Raises exceptions.AttributeError – Please define either input_path or base_path and partition

Warning: Currently only single-line JSON files stored as SequenceFiles or TextFiles are supported!

Note: The init method checks which input parameters are provided and derives the final input_path from them accordingly.

If input_path is not None: Cleans input_path and returns it as the final input_path

Elif base_path and partition are not None: Cleans base_path, infers the sub path from the partition and returns the combined string as the final input_path

Else: Raises an exceptions.AttributeError

extract()
This is the Public API Method to be called for all classes of Extractors

Returns Complex PySpark DataFrame deserialized from the input JSON Files

Return type pyspark.sql.DataFrame

1.3.2 JDBC Source

class JDBCExtractor(*jdbc_options*, *cache=True*)
Bases: *spooq2.extractor.extractor.Extractor*

class JDBCExtractorFullLoad(*query*, *jdbc_options*, *cache=True*)
Bases: *spooq2.extractor.jdbc.JDBCExtractor*

Connects to a JDBC Source and fetches the data defined by the provided Query.

Examples

```
>>> import spooq2.extractor as E
>>>
>>> extractor = E.JDBCExtractorFullLoad(
>>>     query="select id, first_name, last_name, gender, created_at test_db.from␣
↪users",
>>>     jdbc_options={
>>>         "url": "jdbc:postgresql://localhost/test_db",
>>>         "driver": "org.postgresql.Driver",
>>>         "user": "read_only",
>>>         "password": "test123",
>>>     },
>>> )
>>>
>>> extracted_df = extractor.extract()
>>> type(extracted_df)
pyspark.sql.dataframe.DataFrame
```

Parameters

- **query** (str) – Defines the actual query sent to the JDBC Source. This has to be a valid SQL query with respect to the source system (e.g., T-SQL for Microsoft SQL Server).
- **jdbc_options** (dict, optional) –

  A set of parameters to configure the connection to the source:

  - **url** (str) - A JDBC URL of the form jdbc:subprotocol:subname. e.g., jdbc:postgresql://localhost:5432/dbname
  - **driver** (str) - The class name of the JDBC driver to use to connect to this URL.
  - **user** (str) - Username to authenticate with the source database.
  - **password** (str) - Password to authenticate with the source database.

  See pyspark.sql.DataFrameReader.jdbc() and https://spark.apache.org/docs/2.4.3/sql-data-sources-jdbc.html for more information.
- **cache** (bool, defaults to True) – Defines, weather to cache() the dataframe, after it is loaded. Otherwise the Extractor will reload all data from the source system eachtime an action is performed on the DataFrame.

Raises exceptions.AssertionError: – All jdbc_options values need to be present as string variables.

extract()
This is the Public API Method to be called for all classes of Extractors

Returns PySpark dataframe from the input JDBC connection.

Return type pyspark.sql.DataFrame

class JDBCExtractorIncremental(*partition*, *jdbc_options*, *source_table*, *spooq2_values_table*, *spooq2_values_db='spooq2_values'*, *spooq2_values_partition_column='updated_at'*, *cache=True*)
Bases: *spooq2.extractor.jdbc.JDBCExtractor*

Connects to a JDBC Source and fetches the data with respect to boundaries. The boundaries are inferred from the partition to load and logs from previous loads stored in the spooq2_values_table.

Examples

```
>>> import spooq2.extractor as E
>>>
>>> # Boundaries derived from previously logged extractions => ("2020-01-31
↪03:29:59", False)
>>>
>>> extractor = E.JDBCExtractorIncremental(
>>>     partition="20200201",
>>>     jdbc_options={
>>>         "url": "jdbc:postgresql://localhost/test_db",
>>>         "driver": "org.postgresql.Driver",
>>>         "user": "read_only",
>>>         "password": "test123",
>>>     },
>>>     source_table="users",
>>>     spooq2_values_table="spooq2_jdbc_log_users",
>>> )
>>>
>>> extractor._construct_query_for_partition(extractor.partition)
select * from users where updated_at > "2020-01-31 03:29:59"
>>>
>>> extracted_df = extractor.extract()
>>> type(extracted_df)
pyspark.sql.dataframe.DataFrame
```

Parameters

- **partition** (int or str) – Partition to extract. Needed for logging the incremental load in the spooq2_values_table.
- **jdbc_options** (dict, optional) –

  A set of parameters to configure the connection to the source:

  - **url** (str) - A JDBC URL of the form jdbc:subprotocol:subname. e.g., jdbc:postgresql://localhost:5432/dbname
  - **driver** (str) - The class name of the JDBC driver to use to connect to this URL.
  - **user** (str) - Username to authenticate with the source database.
  - **password** (str) - Password to authenticate with the source database.

  See pyspark.sql.DataFrameReader.jdbc() and https://spark.apache.org/docs/2.4.3/sql-data-sources-jdbc.html for more information.
- **source_table** (str) – Defines the tablename of the source to be loaded from. For example 'purchases'. This is necessary to build the query.
- **spooq2_values_table** (str) – Defines the Hive table where previous and future loads of a specific source table are logged. This is necessary to derive boundaries for the current partition.

- **spooq2_values_db** (str, optional) – Defines the Database where the spooq2_values_table is stored. Defaults to *'spooq2_values'*.
- **spooq2_values_partition_column** (str, optional) – The column name which is used for the boundaries. Defaults to *'updated_at'*.
- **cache** (bool, defaults to True) – Defines, weather to cache() the dataframe, after it is loaded. Otherwise the Extractor will reload all data from the source system again, if a second action upon the dataframe is performed.

Raises exceptions.AssertionError: – All jdbc_options values need to be present as string variables.

extract()

Extracts Data from a Source and converts it into a PySpark DataFrame.

Returns

Return type pyspark.sql.DataFrame

Note: This method does not take ANY input parameters. All needed parameters are defined in the initialization of the Extractor Object.

1.3.3 Class Diagram of Extractor Subpackage

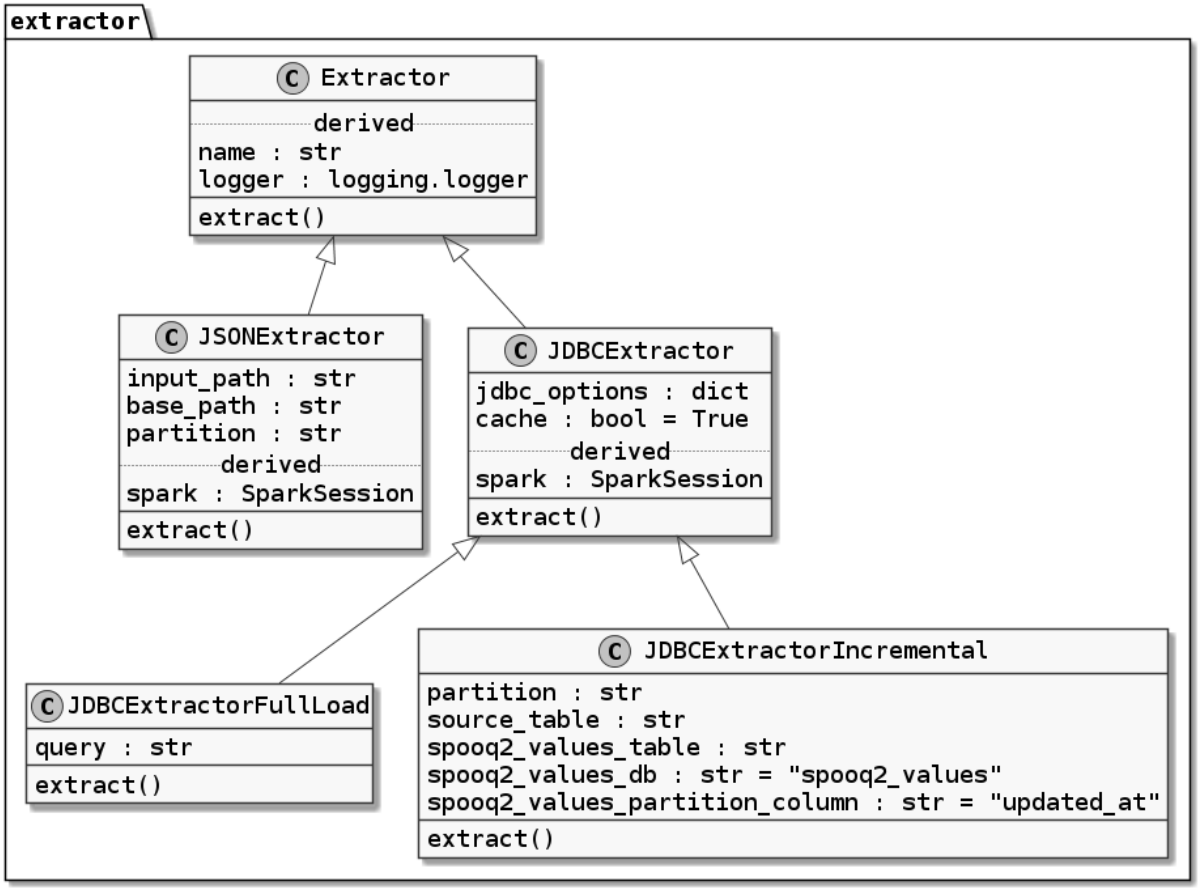


Fig. 1: Class Diagram of Extractor Subpackage

1.3.4 Create your own Extractor

Please see the *Create your own Extractor* for further details.

1.4 Transformers

Transformers take a `pyspark.sql.DataFrame` as an input, transform it accordingly and return a PySpark DataFrame.

Each Transformer class has to have a *transform* method which takes no arguments and returns a PySpark DataFrame.

Possible transformation methods can be **Selecting the most up to date record by id**, **Exploding an array**, **Filter (on an exploded array)**, **Apply basic threshold cleansing** or **Map the incoming DataFrame to at provided structure**.

1.4.1 Exploder

class Exploder(*path_to_array='included'*, *exploded_elem_name='elem'*)

Bases: *spooq2.transformer.transformer.Transformer*

Explodes an array within a DataFrame and drops the column containing the source array.

Examples

```
>>> transformer = Exploder(
>>>     path_to_array="attributes.friends",
>>>     exploded_elem_name="friend",
>>> )
```

Parameters

- **path_to_array** (str, (Defaults to 'included')) – Defines the Column Name / Path to the Array. Dropping nested columns is not supported. Although, you can still explode them.
- **exploded_elem_name** (str, (Defaults to 'elem')) – Defines the column name the exploded column will get. This is important to know how to access the Field afterwards. Writing nested columns is not supported. The output column has to be first level.

Warning: Support for nested column:

path_to_array: PySpark cannot drop a field within a struct. This means the specific field can be referenced and therefore exploded, but not dropped.

exploded_elem_name: If you (re)name a column in the dot notation, is creates a first level column, just with a dot its name. To create a struct with the column as a field you have to redefine the structure or use a UDF.

Note: The `explode()` method of Spark is used internally.

Note: The size of the resulting DataFrame is not guaranteed to be equal to the Input DataFrame!

transform(*input_df*)

Performs a transformation on a DataFrame.

Parameters **input_df** (pyspark.sql.DataFrame) – Input DataFrame

Returns Transformed DataFrame.

Return type pyspark.sql.DataFrame

Note: This method does only take the Input DataFrame as a parameters. All other needed parameters are defined in the initialization of the Transformator Object.

1.4.2 Sieve (Filter)

class Sieve(*filter_expression*)

Bases: *spooq2.transformer.transformer.Transformer*

Filters rows depending on provided filter expression. Only records complying with filter condition are kept.

Examples

```
>>> transformer = T.Sieve(filter_expression=""" attributes.last_name rlike "^.{7}
↪$" """)
```

```
>>> transformer = T.Sieve(filter_expression=""" lower(gender) = "f" """)
```

Parameters **filter_expression** (str) – A valid PySpark SQL expression which returns a boolean

Raises exceptions.ValueError – filter_expression has to be a valid (Spark)SQL expression provided as a string

Note: The `filter()` method is used internally.

Note: The Size of the resulting DataFrame is not guaranteed to be equal to the Input DataFrame!

transform(*input_df*)

Performs a transformation on a DataFrame.

Parameters **input_df** (pyspark.sql.DataFrame) – Input DataFrame

Returns Transformed DataFrame.

Return type pyspark.sql.DataFrame

Note: This method does only take the Input DataFrame as a parameters. All other needed parameters are defined in the initialization of the Transformator Object.

1.4.3 Mapper

Class

class Mapper(*mapping*)

Bases: *spooq2.transformer.transformer.Transformer*

Constructs and applies a PySpark SQL expression, based on the provided mapping.

Examples

```
>>> mapping = [
>>>     ('id',         'data.relationships.food.data.id',   'StringType'),
>>>     ('message_id', 'data.id',                           'StringType'),
>>>     ('type',       'data.relationships.food.data.type', 'StringType'),
>>>     ('created_at', 'elem.attributes.created_at',        'timestamp_ms_to_s'),
>>>     ('updated_at', 'elem.attributes.updated_at',        'timestamp_ms_to_s'),
>>>     ('deleted_at', 'elem.attributes.deleted_at',        'timestamp_ms_to_s'),
>>>     ('brand',      'elem.attributes.brand',             'StringType')
>>> ]
>>> transformer = Mapper(mapping=mapping)
```

```
>>> mapping = [
>>>     ('id',          'data.relationships.food.data.id',   'StringType'),
>>>     ('updated_at',  'elem.attributes.updated_at',        'timestamp_ms_to_s'),
>>>     ('deleted_at',  'elem.attributes.deleted_at',        'timestamp_ms_to_s'),
>>>     ('name',        'elem.attributes.name',              'array')
>>> ]
>>> transformer = Mapper(mapping=mapping)
```

Parameters **mapping** (list of tuple containing three str) – This is the main parameter for this transformation. It essentially gives information about the column names for the output DataFrame, the column names (paths) from the input DataFrame, and their data types. Custom data types are also supported, which can clean, pivot, anonymize, ... the data itself. Please have a look at the *spooq2.transformer.mapper_custom_data_types* module for more information.

Note: Let's talk about Mappings:

The mapping should be a list of tuples which are containing all information per column.

- **Column Name** [str] Sets the name of the column in the resulting output DataFrame.
- **Source Path / Name** [str] Points to the name of the column in the input DataFrame. If the input is a flat DataFrame, it will essentially be the column name. If it is of complex type, it will point to the path of the actual value. For example: `data.relationships.sample.data.id`, where id is the value we want.
- **DataType** [str] DataTypes can be types from `pyspark.sql.types`, selected custom datatypes or injected, ad-hoc custom datatypes. The datatype will be interpreted as a PySpark built-in if it is a member of `pyspark.sql.types`. If it is not an importable PySpark data type, a method to construct the statement will be called by the data type's name.

Note: Please see *spooq2.transformer.mapper_custom_data_types* for all available custom data types and how to inject your own.

Note: Attention: Decimal is NOT SUPPORTED by Hive! Please use Double instead!

---

transform(*input_df*)

Performs a transformation on a DataFrame.

Parameters **input_df** (pyspark.sql.DataFrame) – Input DataFrame

Returns Transformed DataFrame.

Return type pyspark.sql.DataFrame

Note: This method does only take the Input DataFrame as a parameters. All other needed parameters are defined in the initialization of the Transformator Object.

Activity Diagram

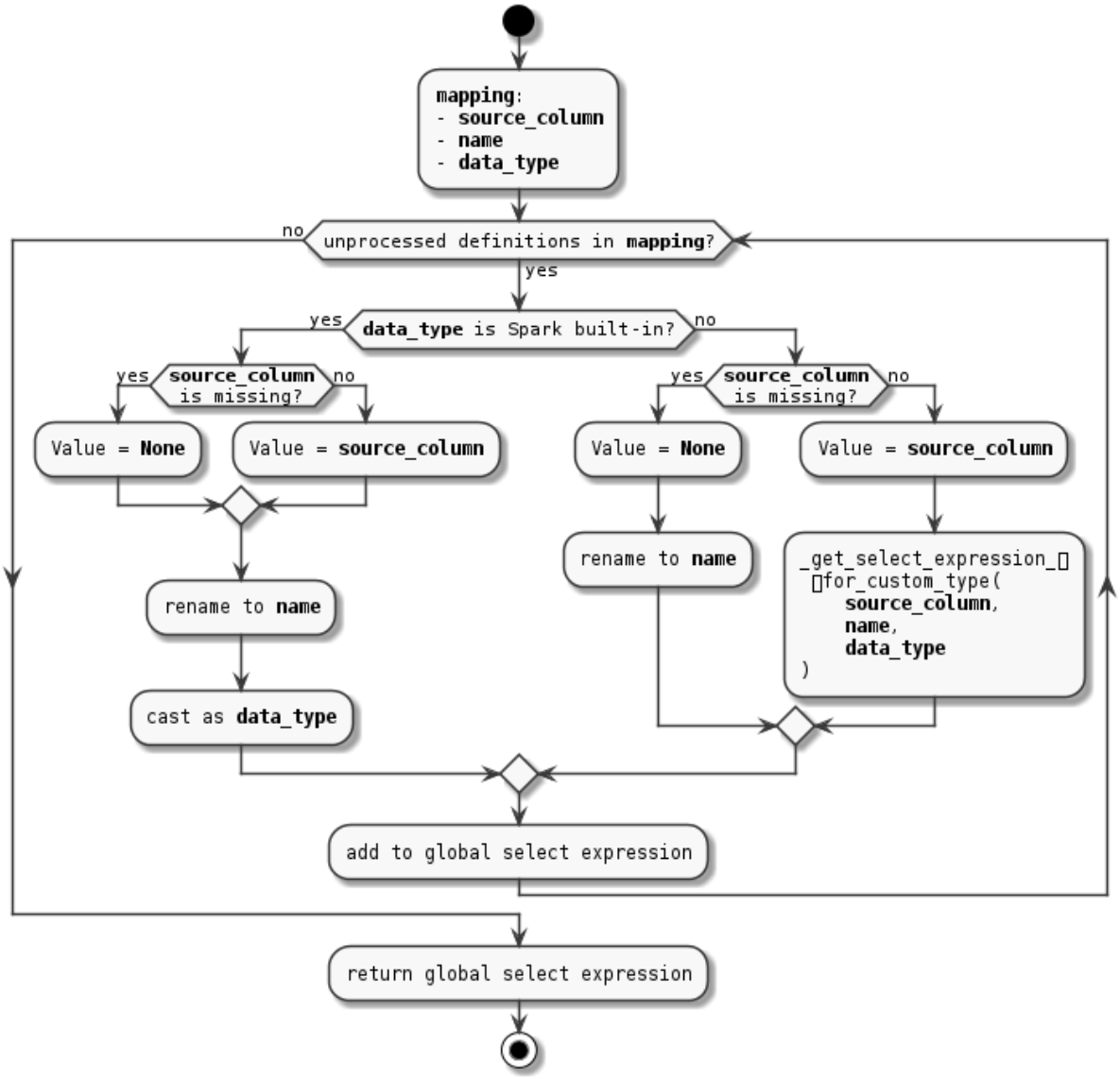


Fig. 2: Activity Diagram for Mapper Transformer

Custom Mapping Methods

This is a collection of module level methods to construct a specific PySpark DataFrame query for custom defined data types.

These methods are not meant to be called directly but via the the *Mapper* transformer. Please see that particular class on how to apply custom data types.

For injecting your **own custom data types**, please have a visit to the *add_custom_data_type()* method!

add_custom_data_type (*function_name*, *func*)

Registers a custom data type in runtime to be used with the *Mapper* transformer.

Example

```
>>> import spooq2.transformer.mapper_custom_data_types as custom_types
>>> import spooq2.transformer as T
>>> from pyspark.sql import Row, functions as F, types as sql_types
```

```
>>> def hello_world(source_column, name):
>>>     "A UDF (User Defined Function) in Python"
>>>     def _to_hello_world(col):
>>>         if not col:
>>>             return None
>>>         else:
>>>             return "Hello World"
>>>
>>>     udf_hello_world = F.udf(_to_hello_world, sql_types.StringType())
>>>     return udf_hello_world(source_column).alias(name)
>>>
>>> input_df = spark.createDataFrame(
>>>     [Row(hello_from=u'tsivorn1@who.int'),
>>>      Row(hello_from=u''),
>>>      Row(hello_from=u'gisaksen4@skype.com')]
>>> )
>>>
>>> custom_types.add_custom_data_type(function_name="hello_world", func=hello_
↪world)
>>> transformer = T.Mapper(mapping=[("hello_who", "hello_from", "hello_world")])
>>> df = transformer.transform(input_df)
>>> df.show()
+-----------+
|  hello_who|
+-----------+
|Hello World|
|       null|
|Hello World|
+-----------+
```

```
>>> def first_and_last_name(source_column, name):
>>>     "A PySpark SQL expression referencing multiple columns"
>>>     return F.concat_ws("_", source_column, F.col("attributes.last_name")).
↪alias(name)
>>>
>>> custom_types.add_custom_data_type(function_name="full_name", func=first_and_
↪last_name)
>>>
>>> transformer = T.Mapper(mapping=[
>>>     ("first_name", "attributes.first_name", "StringType"),
>>>     ("last_name",  "attributes.last_name",  "StringType"),
>>>     ("full_name",  "attributes.first_name", "full_name"),
>>> ])
```

Parameters

- **function_name** (`str`) – The name of your custom data type
- **func** (*compatible function*) – The PySpark dataframe function which will be called on a column, defined in the mapping of the Mapper class. Required input parameters are `source_column` and `name`. Please see the note about required input parameter of custom data types for more information!

Note: Required input parameter of custom data types:

source_column (`pyspark.sql.Column`) - This is where your logic will be applied. The mapper transformer takes care of calling this method with the right column so you can just handle it like an object which you would get from `df["some_attribute"]`.

name (`str`) - The name how the resulting column will be named. Nested attributes are not supported. The Mapper transformer takes care of calling this method with the right column name.

_get_select_expression_for_custom_type (*source_column*, *name*, *data_type*)

Internal method for calling functions dynamically

_generate_select_expression_for_as_is(*source_column*, *name*)
alias for *_generate_select_expression_without_casting*

_generate_select_expression_for_keep(*source_column*, *name*)
alias for *_generate_select_expression_without_casting*

_generate_select_expression_for_no_change(*source_column*, *name*)
alias for *_generate_select_expression_without_casting*

_generate_select_expression_without_casting(*source_column*, *name*)
Returns a column without casting. This is especially useful if you need to keep a complex data type, like an array, list or a struct.

```
>>> from spooq2.transformer import Mapper
>>>
>>> input_df.head(3)
[Row(friends=[Row(first_name=None, id=3993, last_name=None), Row(first_name=u'Ruò
↪', id=17484, last_name=u'Trank')]),
 Row(friends=[]),
 Row(friends=[Row(first_name=u'Daphnée', id=16707, last_name=u'Lyddiard'),␣
↪Row(first_name=u'Adélaïde', id=17429, last_name=u'Wisdom')])]
>>> mapping = [("my_friends", "friends", "as_is")]
>>> output_df = Mapper(mapping).transform(input_df)
>>> output_df.head(3)
[Row(my_friends=[Row(first_name=None, id=3993, last_name=None), Row(first_name=u
↪'Ruò', id=17484, last_name=u'Trank')]),
 Row(my_friends=[]),
 Row(my_friends=[Row(first_name=u'Daphnée', id=16707, last_name=u'Lyddiard'),␣
↪Row(first_name=u'Adélaïde', id=17429, last_name=u'Wisdom')])]
```

_generate_select_expression_for_json_string(*source_column*, *name*)
Returns a column as json compatible string. Nested hierarchies are supported. The unicode representation of a column will be returned if an error occurs.

Example

```
>>> from spooq2.transformer import Mapper
>>>
>>> input_df.head(3)
[Row(friends=[Row(first_name=None, id=3993, last_name=None), Row(first_name=u'Ruò
↪', id=17484, last_name=u'Trank')]),
 Row(friends=[]),
 Row(friends=[Row(first_name=u'Daphnée', id=16707, last_name=u'Lyddiard'),␣
↪Row(first_name=u'Adélaïde', id=17429, last_name=u'Wisdom')])]    >>> mapping =␣
↪[("friends_json", "friends", "json_string")]
>>> mapping = [("friends_json", "friends", "json_string")]
>>> output_df = Mapper(mapping).transform(input_df)
>>> output_df.head(3)
[Row(friends_json=u'[{"first_name": null, "last_name": null, "id": 3993}, {
↪"first_name": "Ru\u00f2", "last_name": "Trank", "id": 17484}]'),
 Row(friends_json=None),
 Row(friends_json=u'[{"first_name": "Daphn\u00e9e", "last_name": "Lyddiard", "id
↪": 16707}, {"first_name": "Ad\u00e9la\u00efde", "last_name": "Wisdom", "id":␣
↪17429}]')]
```

_generate_select_expression_for_timestamp_ms_to_ms(*source_column*, *name*)
This Constructor is used for unix timestamps. The values are cleaned next to casting and renaming. If the values are not between *01.01.1970* and *31.12.2099*, NULL will be returned. Cast to `pyspark.sql.types.LongType`

Example

```
>>> from pyspark.sql import Row
>>> from spooq2.transformer import Mapper
>>>
>>> input_df = spark.createDataFrame([
>>>     Row(time_sec=1581540839000),   # 02/12/2020 @ 8:53pm (UTC)
>>>     Row(time_sec=-4887839000),     # Invalid!
>>>     Row(time_sec=4737139200000)    # 02/12/2120 @ 12:00am (UTC)
>>> ])
>>>
>>> mapping = [("unix_ts", "time_sec", "timestamp_ms_to_ms")]
>>> output_df = Mapper(mapping).transform(input_df)
>>> output_df.head(3)
[Row(unix_ts=1581540839000), Row(unix_ts=None), Row(unix_ts=None)]
```

Note: *input* in **milli seconds** *output* in **milli seconds**

_generate_select_expression_for_timestamp_ms_to_s(*source_column*, *name*)

This Constructor is used for unix timestamps. The values are cleaned next to casting and renaming. If the values are not between *01.01.1970* and *31.12.2099*, NULL will be returned. Cast to `pyspark.sql.types.LongType`

Example

```
>>> from pyspark.sql import Row
>>> from spooq2.transformer import Mapper
>>>
>>> input_df = spark.createDataFrame([
>>>     Row(time_sec=1581540839000),   # 02/12/2020 @ 8:53pm (UTC)
>>>     Row(time_sec=-4887839000),     # Invalid!
>>>     Row(time_sec=4737139200000)    # 02/12/2120 @ 12:00am (UTC)
>>> ])
>>>
>>> mapping = [("unix_ts", "time_sec", "timestamp_ms_to_s")]
>>> output_df = Mapper(mapping).transform(input_df)
>>> output_df.head(3)
[Row(unix_ts=1581540839), Row(unix_ts=None), Row(unix_ts=None)]
```

Note: *input* in **milli seconds** *output* in **seconds**

_generate_select_expression_for_timestamp_s_to_ms(*source_column*, *name*)

This Constructor is used for unix timestamps. The values are cleaned next to casting and renaming. If the values are not between *01.01.1970* and *31.12.2099*, NULL will be returned. Cast to `pyspark.sql.types.LongType`

Example

```
>>> from pyspark.sql import Row
>>> from spooq2.transformer import Mapper
>>>
>>> input_df = spark.createDataFrame([
>>>     Row(time_sec=1581540839),   # 02/12/2020 @ 8:53pm (UTC)
>>>     Row(time_sec=-4887839),     # Invalid!
>>>     Row(time_sec=4737139200)    # 02/12/2120 @ 12:00am (UTC)
>>> ])
>>>
>>> mapping = [("unix_ts", "time_sec", "timestamp_s_to_ms")]
```

(continues on next page)

(continued from previous page)

```
>>> output_df = Mapper(mapping).transform(input_df)
>>> output_df.head(3)
[Row(unix_ts=1581540839000), Row(unix_ts=None), Row(unix_ts=None)]
```

Note: *input* in **seconds** *output* in **milli seconds**

_generate_select_expression_for_timestamp_s_to_s(*source_column*, *name*)

This Constructor is used for unix timestamps. The values are cleaned next to casting and renaming. If the values are not between *01.01.1970* and *31.12.2099*, NULL will be returned. Cast to `pyspark.sql.types.LongType`

Example

```
>>> from pyspark.sql import Row
>>> from spooq2.transformer import Mapper
>>>
>>> input_df = spark.createDataFrame([
>>>     Row(time_sec=1581540839),   # 02/12/2020 @ 8:53pm (UTC)
>>>     Row(time_sec=-4887839),     # Invalid!
>>>     Row(time_sec=4737139200)    # 02/12/2120 @ 12:00am (UTC)
>>> ])
>>>
>>> mapping = [("unix_ts", "time_sec", "timestamp_s_to_ms")]
>>> output_df = Mapper(mapping).transform(input_df)
>>> output_df.head(3)
[Row(unix_ts=1581540839), Row(unix_ts=None), Row(unix_ts=None)]
```

Note: *input* in **seconds** *output* in **seconds**

_generate_select_expression_for_StringNull(*source_column*, *name*)

Used for Anonymizing. Input values will be ignored and replaced by NULL, Cast to `pyspark.sql.types.StringType`

Example

```
>>> from pyspark.sql import Row
>>> from spooq2.transformer import Mapper
>>>
>>> input_df = spark.createDataFrame(
>>>     [Row(email=u'tsivorn1@who.int'),
>>>      Row(email=u''),
>>>      Row(email=u'gisaksen4@skype.com')]
>>> )
>>>
>>> mapping = [("email", "email", "StringNull")]
>>> output_df = Mapper(mapping).transform(input_df)
>>> output_df.head(3)
[Row(email=None), Row(email=None), Row(email=None)]
```

_generate_select_expression_for_IntNull(*source_column*, *name*)

Used for Anonymizing. Input values will be ignored and replaced by NULL, Cast to `pyspark.sql.types.IntegerType`

Example

```
>>> from pyspark.sql import Row
>>> from spooq2.transformer import Mapper
>>>
>>> input_df = spark.createDataFrame(
>>>     [Row(facebook_id=3047288),
>>>      Row(facebook_id=0),
>>>      Row(facebook_id=57815)]
>>> )
>>>
>>> mapping = [("facebook_id", "facebook_id", "IntNull")]
>>> output_df = Mapper(mapping).transform(input_df)
>>> output_df.head(3)
[Row(facebook_id=None), Row(facebook_id=None), Row(facebook_id=None)]
```

_generate_select_expression_for_StringBoolean(*source_column*, *name*)

Used for Anonymizing. The column's value will be replaced by *"1"* if it is:

- not NULL and
- not an empty string

Example

```
>>> from pyspark.sql import Row
>>> from spooq2.transformer import Mapper
>>>
>>> input_df = spark.createDataFrame(
>>>     [Row(email=u'tsivorn1@who.int'),
>>>      Row(email=u''),
>>>      Row(email=u'gisaksen4@skype.com')]
>>> )
>>>
>>> mapping = [("email", "email", "StringBoolean")]
>>> output_df = Mapper(mapping).transform(input_df)
>>> output_df.head(3)
[Row(email=u'1'), Row(email=None), Row(email=u'1')]
```

_generate_select_expression_for_IntBoolean(*source_column*, *name*)

Used for Anonymizing. The column's value will be replaced by *1* if it contains a non-NULL value.

Example

```
>>> from pyspark.sql import Row
>>> from spooq2.transformer import Mapper
>>>
>>> input_df = spark.createDataFrame(
>>>     [Row(facebook_id=3047288),
>>>      Row(facebook_id=0),
>>>      Row(facebook_id=None)]
>>> )
>>>
>>> mapping = [("facebook_id", "facebook_id", "IntBoolean")]
>>> output_df = Mapper(mapping).transform(input_df)
>>> output_df.head(3)
[Row(facebook_id=1), Row(facebook_id=1), Row(facebook_id=None)]
```

Note: *0* (zero) or negative numbers are still considered as valid values and therefore converted to *1*.

_generate_select_expression_for_TimestampMonth(*source_column*, *name*)

Used for Anonymizing. Can be used to keep the age but obscure the explicit birthday. This custom datatype requires a `pyspark.sql.types.TimestampType` column as input. The datetime value will be set to the first day of the month.

Example

```
>>> from pyspark.sql import Row
>>> from datetime import datetime
>>> from spooq2.transformer import Mapper
>>>
>>> input_df = spark.createDataFrame(
>>>     [Row(birthday=datetime(2019, 2, 9, 2, 45)),
>>>      Row(birthday=None),
>>>      Row(birthday=datetime(1988, 1, 31, 8))]
>>> )
>>>
>>> mapping = [("birthday", "birthday", "TimestampMonth")]
>>> output_df = Mapper(mapping).transform(input_df)
>>> output_df.head(3)
[Row(birthday=datetime.datetime(2019, 2, 1, 0, 0)),
 Row(birthday=None),
 Row(birthday=datetime.datetime(1988, 1, 1, 0, 0))]
```

1.4.4 Threshold-based Cleaner

class ThresholdCleaner(*thresholds={}*)

Bases: *spooq2.transformer.transformer.Transformer*

Sets outiers within a DataFrame to a default value. Takes a dictionary with valid value ranges for each column to be cleaned.

Example

```
>>> transformer = ThresholdCleaner(
>>>     thresholds={
>>>         "created_at": {
>>>             "min": 0,
>>>             "max": 1580737513,
>>>             "default": None
>>>         },
>>>         "size_cm": {
>>>             "min": 70,
>>>             "max": 250,
>>>             "default": None
>>>         },
>>>     }
>>> )
```

Parameters **thresholds** (`dict`) – Dictionary containing column names and respective valid ranges

Returns The transformed DataFrame

Return type `pyspark.sql.DataFrame`

Raises `exceptions.ValueError` – Threshold-based cleaning only supports Numeric Types! Column of name: {col_name} and type of: {col_type} was provided

Warning: Only numeric data types are supported!

transform(*input_df*)
Performs a transformation on a DataFrame.

Parameters **input_df** (pyspark.sql.DataFrame) – Input DataFrame

Returns Transformed DataFrame.

Return type pyspark.sql.DataFrame

Note: This method does only take the Input DataFrame as a parameters. All other needed parameters are defined in the initialization of the Transformator Object.

1.4.5 Newest by Group (Most current record per ID)

class NewestByGroup(*group_by=['id'], order_by=['updated_at', 'deleted_at']*)
Bases: *spooq2.transformer.transformer.Transformer*

Groups, orders and selects first element per group.

Example

```
>>> transformer = NewestByGroup(
>>>     group_by=["first_name", "last_name"],
>>>     order_by=["created_at_ms", "version"]
>>> )
```

Parameters

- **group_by** (str or list of str, (Defaults to ['id'])) – List of attributes to be used within the Window Function as Grouping Arguments.
- **order_by** (str or list of str, (Defaults to ['updated_at', 'deleted_at'])) – List of attributes to be used within the Window Function as Ordering Arguments. All columns will be sorted in **descending** order.

Raises exceptions.AttributeError – If any Attribute in group_by or order_by is not contained in the input DataFrame.

Note: PySpark's Window function is used internally The first row (row_number()) per window will be selected and returned.

transform(*input_df*)
Performs a transformation on a DataFrame.

Parameters **input_df** (pyspark.sql.DataFrame) – Input DataFrame

Returns Transformed DataFrame.

Return type pyspark.sql.DataFrame

Note: This method does only take the Input DataFrame as a parameters. All other needed parameters are defined in the initialization of the Transformator Object.

1.4.6 Class Diagram of Transformer Subpackage

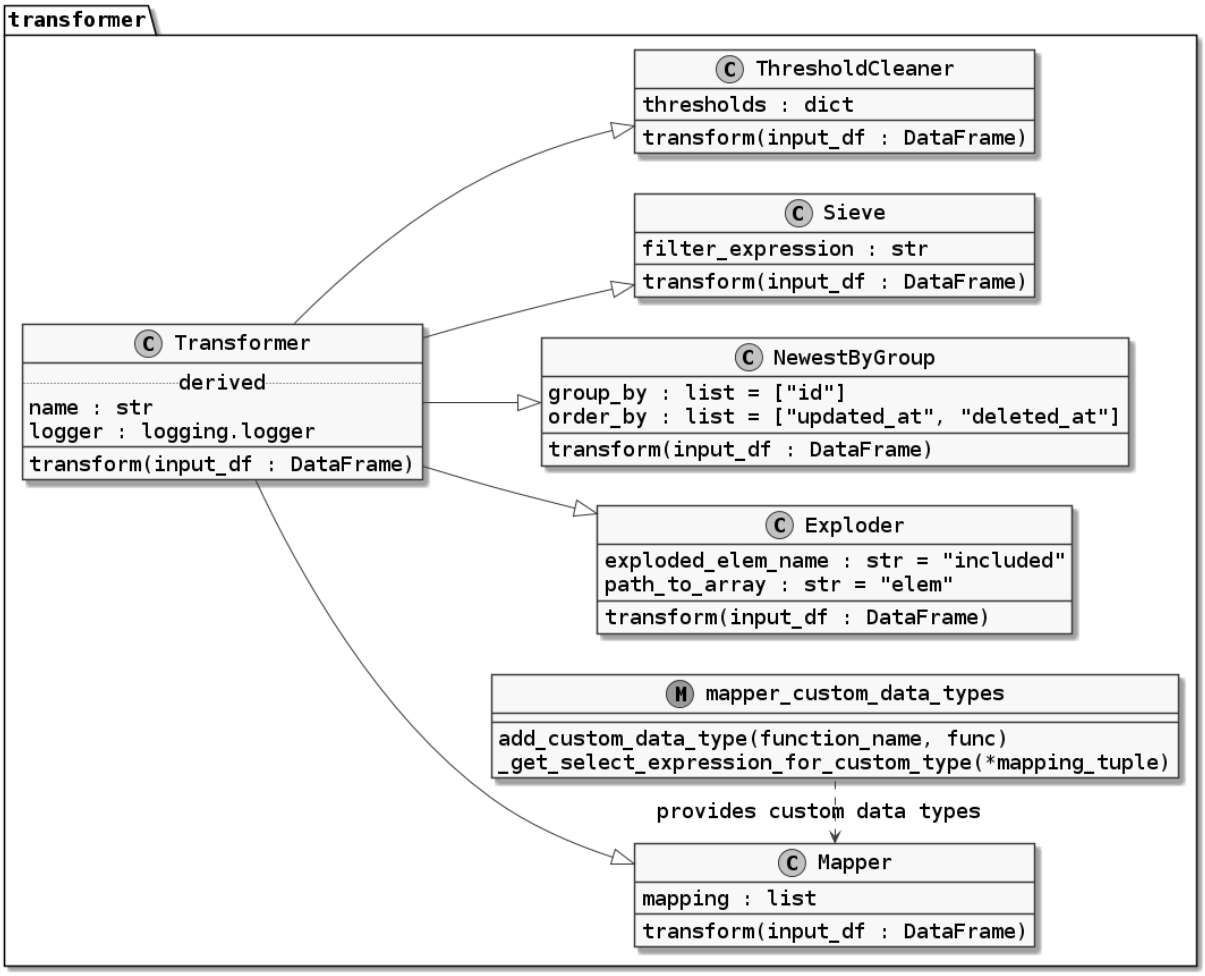


Fig. 3: Class Diagram of Transformer Subpackage

1.4.7 Create your own Transformer

Please see the *Create your own Transformer* for further details.

1.5 Loaders

Loaders take a `pyspark.sql.DataFrame` as an input and save it to a sink.

Each Loader class has to have a *load* method which takes a DataFrame as single paremter.

Possible Loader sinks can be **Hive Tables**, **Kudu Tables**, **HBase Tables**, **JDBC Sinks** or **ParquetFiles**.

1.5.1 Hive Database

class HiveLoader(*db_name, table_name, partition_definitions=[{'default_value': None, 'column_type': 'IntegerType', 'column_name': 'dt'}], clear_partition=True, repartition_size=40, auto_create_table=True, overwrite_partition_value=True*)

Bases: *spooq2.loader.loader.Loader*

Persists a PySpark DataFrame into a Hive Table.

Examples

```
>>> HiveLoader(
>>>     db_name="users_and_friends",
>>>     table_name="friends_partitioned",
>>>     partition_definitions=[{
>>>         "column_name": "dt",
>>>         "column_type": "IntegerType",
>>>         "default_value": 20200201}],
>>>     clear_partition=True,
>>>     repartition_size=10,
>>>     overwrite_partition_value=False,
>>>     auto_create_table=False,
>>> ).load(input_df)
```

```
>>> HiveLoader(
>>>     db_name="users_and_friends",
>>>     table_name="all_friends",
>>>     partition_definitions=[],
>>>     repartition_size=200,
>>>     auto_create_table=True,
>>> ).load(input_df)
```

Parameters

- **db_name** (str) – The database name to load the data into.
- **table_name** (str) – The table name to load the data into. The database name must not be included in this parameter as it is already defined in the *db_name* parameter.
- **partition_definitions** (list of dict) – (Defaults to *[{"column_name": "dt", "column_type": "IntegerType", "default_value": None}]*).
  - **column_name** (str) - The Column's Name to partition by.
  - **column_type** (str) - The PySpark SQL DataType for the Partition Value as a String. This should normally either be 'IntegerType()' or 'StringType()'
  - **default_value** (str or int) - If *column_name* does not contain a value or *overwrite_partition_value* is set, this value will be used for the partitioning
- **clear_partition** (bool, (Defaults to True)) – This flag tells the Loader to delete the defined partitions before inserting the input DataFrame into the target table. Has no effect if no partitions are defined.
- **repartition_size** (int, (Defaults to 40)) – The DataFrame will be repartitioned on Spark level before inserting into the table. This effects the number of output files on which the Hive table is based.
- **auto_create_table** (bool, (Defaults to True)) – Whether the target table will be created if it does not yet exist.
- **overwrite_partition_value** (bool, (Defaults to True)) – Defines whether the values of columns defined in *partition_definitions* should explicitly set by default_values.

Raises

- exceptions.AssertionError: – partition_definitions has to be a list containing dicts. Expected dict content: 'column_name', 'column_type', 'default_value' per partition_definitions item.
- exceptions.AssertionError: – Items of partition_definitions have to be dictionaries.
- exceptions.AssertionError: – No column name set!
- exceptions.AssertionError: – Not a valid (PySpark) datatype for the partition column {name} | {type}.

- exceptions.AssertionError: – *clear_partition* is only supported if *overwrite_partition_value* is also enabled. This would otherwise result in clearing partitions on basis of dynamically values (from DataFrame) instead of explicitly defining the partition(s) to clear.

load(*input_df*)

Persists data from a PySpark DataFrame to a target table.

Parameters **input_df** (pyspark.sql.DataFrame) – Input DataFrame which has to be loaded to a target destination.

Note: This method takes only a single DataFrame as an input parameter. All other needed parameters are defined in the initialization of the Loader object.

Activity Diagram

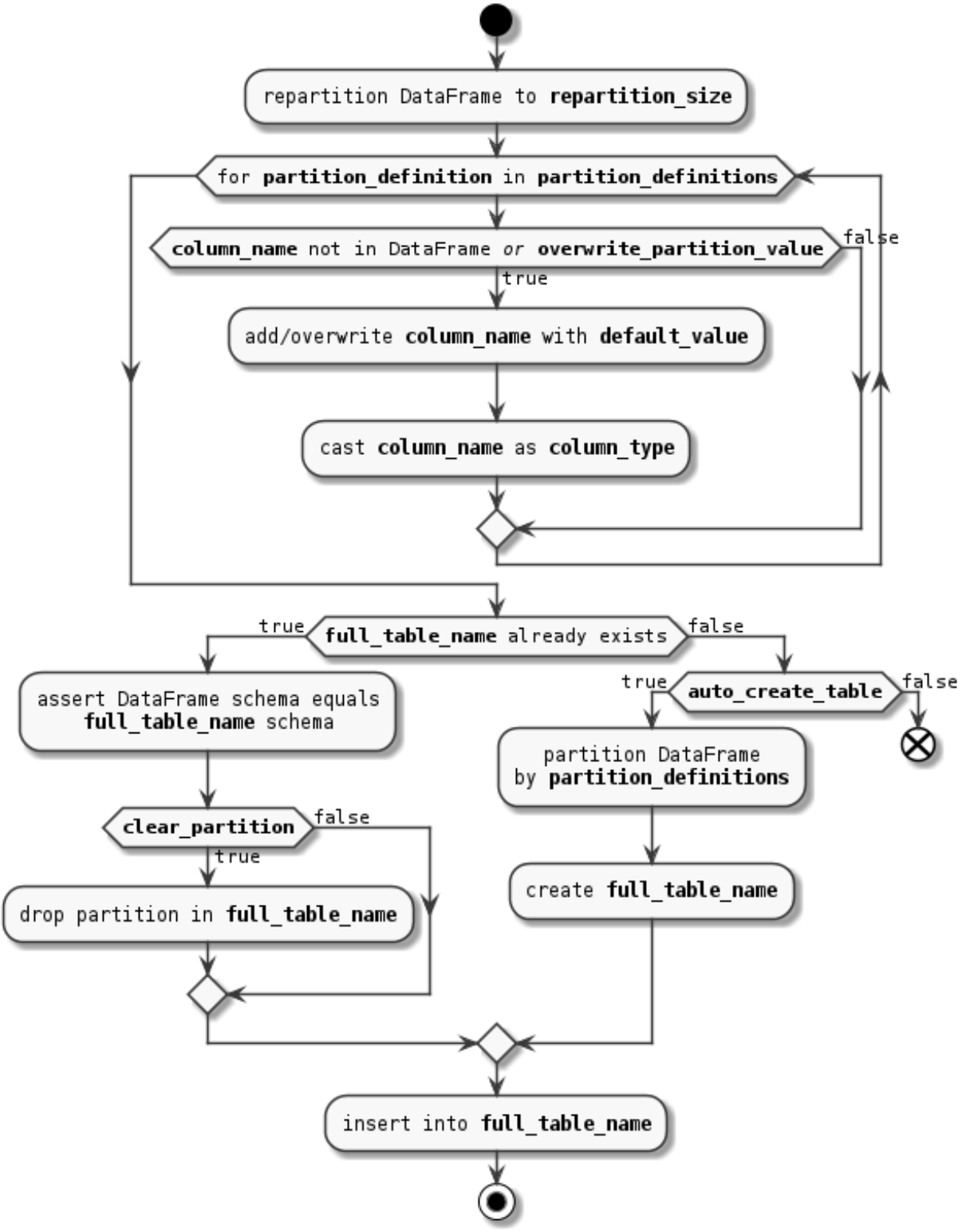


Fig. 4: Activity Diagram for Hive Loader

1.5.2 Class Diagram of Loader Subpackage

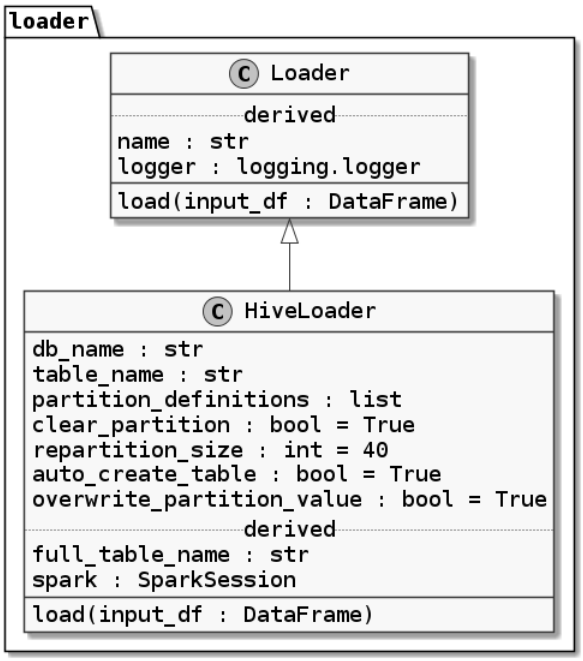


Fig. 5: Class Diagram of Loader Subpackage

1.5.3 Create your own Loader

Please see the *Create your own Loader* for further details.

1.6 Pipeline

1.6.1 Pipeline

This type of object glues the aforementioned processes together and extracts, transforms (Transformer chain possible) and loads the data from start to end.

class Pipeline(*input_df=None*, *bypass_loader=False*)

Bases: `object`

Represents a Pipeline of an Extractor, (multiple) Transformers and a Loader Object.

extractor

The entry point of the Pipeline. Extracts a DataFrame from a Source.

Type Subclass of `spooq2.extractor.Extractor`

transformers

The Data Wrangling Part of the Pipeline. A chain of Transformers, a single Transformer or a PassThrough Transformer can be set and used.

Type List of Subclasses of `spooq2.transformer.Transformer` Objects

loader

The exit point of the Pipeline. Loads a DataFrame to a target Sink.

Type Subclass of `spooq2.loader.Loader`

name

Sets the *__name__* of the class' type as *name*, which is essentially the Class' Name.

Type `str`

logger
Shared, class level logger for all instances.

Type logging.Logger

Example

```
>>> from spooq2.pipeline import Pipeline
>>> import spooq2.extractor as   E
>>> import spooq2.transformer as T
>>> import spooq2.loader as      L
>>>
>>> #  Definition how the output table should look like and where the attributes␣
↪come from:
>>> users_mapping = [
>>>     ("id",              "id",                    "IntegerType"),
>>>     ("guid",            "guid",                  "StringType"),
>>>     ("forename",        "attributes.first_name", "StringType"),
>>>     ("surename",        "attributes.last_name",  "StringType"),
>>>     ("gender",          "attributes.gender",     "StringType"),
>>>     ("has_email",       "attributes.email",      "StringBoolean"),
>>>     ("has_university",  "attributes.university", "StringBoolean"),
>>>     ("created_at",      "meta.created_at_ms",    "timestamp_ms_to_s"),
>>> ]
>>>
>>> #  The main object where all steps are defined:
>>> users_pipeline = Pipeline()
>>>
>>> #  Defining the EXTRACTION:
>>> users_pipeline.set_extractor(E.JSONExtractor(
>>>     input_path="tests/data/schema_v1/sequenceFiles"
>>> ))
>>>
>>> #  Defining the TRANSFORMATION:
>>> users_pipeline.add_transformers([
>>>     T.Mapper(mapping=users_mapping),
>>>     T.ThresholdCleaner(thresholds={"created_at": {
>>>                                                   "min": 0,
>>>                                                   "max": 1580737513,
>>>                                                   "default": None}}),
>>>     T.NewestByGroup(group_by="id", order_by="created_at")
>>> ])
>>>
>>> #  Defining the LOAD:
>>> users_pipeline.set_loader(L.HiveLoader(
>>>     db_name="users_and_friends",
>>>     table_name="users",
>>>     partition_definitions=[{
>>>         "column_name": "dt",
>>>         "column_type": "IntegerType",
>>>         "default_value": 20200201}],
>>>     repartition_size=10,
>>> ))
>>>
>>> #  Executing the whole ETL pipeline
>>> users_pipeline.execute()
```

execute()
Executes the whole Pipeline at once.

Extracts from the Source, transformes the DataFrame and loads it into a target Sink.

Returns **input_df** – **If** the bypass_loader attribute was set to True in the Pipeline class, the output DataFrame from the Transformer(s) will be directly returned.

Return type pyspark.sql.DataFrame

Note: This method does not take ANY input parameters. All needed parameters are defined at the initialization phase.

extract()
Calls the extract Method on the Extractor Object.

Returns The output_df from the Extractor used as the input for the Transformer (chain).

Return type pyspark.sql.DataFrame

transform(*input_df*)
Calls the transform Method on the Transformer Object(s) in the order of importing the Objects while passing the DataFrame.

Parameters **input_df** (pyspark.sql.DataFrame) – The output DataFrame of the Extractor Object.

Returns The input DataFrame for the Loader.

Return type pyspark.sql.DataFrame

load(*input_df*)
Calls the load Method on the Loader Object.

Parameters **input_df** (pyspark.sql.DataFrame) – The output DataFrame from the Transformer(s).

Returns **input_df** – **If** the bypass_loader attribute was set to True in the Pipeline class, the output DataFrame from the Transformer(s) will be directly returned.

Return type pyspark.sql.DataFrame

set_extractor(*extractor*)
Sets an Extractor Object to be used within the Pipeline.

Parameters **extractor** (Subclass of spooq2.extractor.Extractor) – An already initialized Object of any Subclass of spooq2.extractor.Extractor.

Raises exceptions.AssertionError: – An input_df was already provided which bypasses the extraction action

add_transformers(*transformers*)
Adds a list of Transformer Objects to be used within the Pipeline.

Parameters **transformer** (list of Subclass of spooq2.transformer.Transformer) – Already initialized Object of any Subclass of spooq2.transformer.Transformer.

clear_transformers()
Clears the list of already added Transformers.

set_loader(*loader*)
Sets an Loader Object to be used within the Pipeline.

Parameters **loader** (Subclass of spooq2.loader.Loader) – An already initialized Object of any Subclass of spooq2.loader.Loader.

Raises exceptions.AssertionError: – You can not set a loader if the *bypass_loader* parameter is set.

1.6.2 Pipeline Factory

To decrease the complexity of building data pipelines for data engineers, an expert system or business rules engine can be used to automatically build and configure a data pipeline based on context variables, groomed metadata, and relevant rules.

class PipelineFactory(*url='http://localhost:5000/pipeline/get'*)

Bases: object

Provides an interface to automatically construct pipelines for Spooq.

Example

```
>>> pipeline_factory = PipelineFactory()
>>>
>>> #  Fetch user data set with applied mapping, filtering,
>>> #  and cleaning transformers
>>> df = pipeline_factory.execute({
>>>       "entity_type": "user",
>>>       "date": "2018-10-20",
>>>       "time_range": "last_day"})
>>>
>>> #  Load user data partition with applied mapping, filtering,
>>> #  and cleaning transformers to a hive database
>>> pipeline_factory.execute({
>>>       "entity_type": "user",
>>>       "date": "2018-10-20",
>>>       "batch_size": "daily"})
```

url

The end point of an expert system which will be called to infer names and parameters.

Type str, (Defaults to "http://localhost:5000/pipeline/get")

Note: PipelineFactory is only responsible for querying an expert system with provided parameters and constructing a Spooq pipeline out of the response. It does not have any reasoning capabilities itself! It requires therefore a HTTP service responding with a JSON object containing following structure:

```
{
    "extractor": {"name": "Type1Extractor", "params": {"key 1": "val 1", "key N
↪": "val N"}},
    "transformers": [
        {"name": "Type1Transformer", "params": {"key 1": "val 1", "key N": "val N
↪"}},
        {"name": "Type2Transformer", "params": {"key 1": "val 1", "key N": "val N
↪"}},
        {"name": "Type3Transformer", "params": {"key 1": "val 1", "key N": "val N
↪"}},
        {"name": "Type4Transformer", "params": {"key 1": "val 1", "key N": "val N
↪"}},
        {"name": "Type5Transformer", "params": {"key 1": "val 1", "key N": "val N
↪"}},
    ],
    "loader": {"name": "Type1Loader", "params": {"key 1": "val 1", "key N": "val␣
↪N"}}
}
```

Hint: There is an experimental implementation of an expert system which complies with the requirements of PipelineFactory called *spooq_rules*. If you are interested, please ask the author of Spooq about it.

---

execute(*context_variables*)

Fetches a ready-to-go pipeline instance via *get_pipeline()* and executes it.

Parameters **context_variables** (dict) – These collection of parameters should describe the current context about the use case of the pipeline. Please see the examples of the PipelineFactory class' documentation.

Returns

- `pyspark.sql.DataFrame` – If the loader component is by-passed (in the case of ad_hoc use cases).
- `None` – If the loader component does not return a value (in the case of persisting data).

get_metadata(*context_variables*)
Sends a POST request to the defined endpoint (*url*) containing the supplied context variables.

Parameters **context_variables** (`dict`) – These collection of parameters should describe the current context about the use case of the pipeline. Please see the examples of the PipelineFactory class' documentation.

Returns Names and parameters of each ETL component to construct a Spooq pipeline

Return type `dict`

get_pipeline(*context_variables*)
Fetches the necessary metadata via *`get_metadata()`* and returns a ready-to-go pipeline instance.

Parameters **context_variables** (`dict`) – These collection of parameters should describe the current context about the use case of the pipeline. Please see the examples of the PipelineFactory class' documentation.

Returns A Spooq pipeline instance which is fully configured and can still be adapted and consequently executed.

Return type `Pipeline`

1.6.3 Class Diagram of Pipeline Subpackage

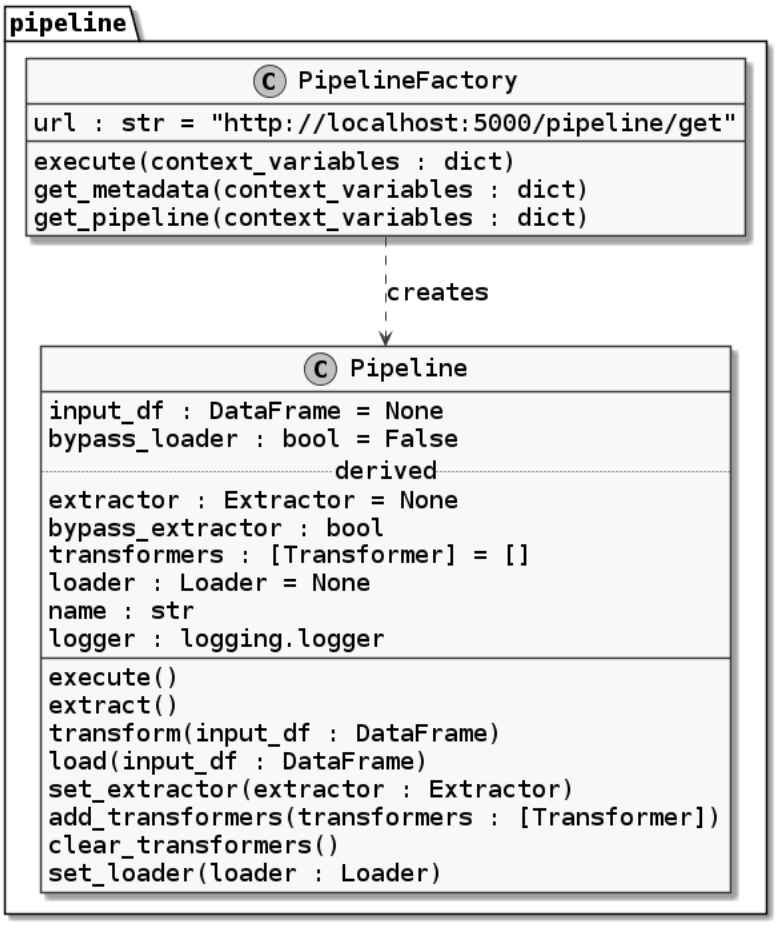


Fig. 6: Class Diagram of Pipeline Subpackage

1.7 Spooq Base

1.7.1 Global Logger

Global Logger instance used by Spooq2.

Example

```
>>> import logging
>>> logga = logging.getLogger("spooq2")
<logging.Logger at 0x7f5dc8eb2890>
>>> logga.info("Hello World")
[spooq2] 2020-03-21 23:55:48,253 INFO logging_example::<module>::4: Hello World
```

initialize()
Initializes the global logger for Spooq with pre-defined levels for `stdout` and `stderr`. No input parameters are needed, as the configuration is received via *get_logging_level()*.

Note:

The output format is defined as:

"[%(name)s] %(asctime)s %(levelname)s %(module)s::%(funcName)s::%(lineno)d: %(message)s"
For example "[spooq2] 2020-03-11 15:40:59,313 DEBUG newest_by_group::__init__::53: group by columns: [u'user_id']"

> **Warning:** The `root` logger of python is also affected as it has to have a level at least as fine grained as the logger of Spooq, to be able to produce an output.

get_logging_level()
Returns the logging level depending on the environment variable *SPOOQ_ENV*.

Note:

If SPOOQ_ENV is

- **dev** -> "DEBUG"
- **test** -> "ERROR"
- something else -> "INFO"

Returns Logging level

Return type `str`

1.7.2 Extractor Base Class

Extractors are used to fetch, extract and convert a source data set into a PySpark DataFrame. Exemplary extraction sources are **JSON Files** on file systems like HDFS, DBFS or EXT4 and relational database systems via **JDBC**.

class Extractor
Bases: `object`

Base Class of Extractor Classes.

name
Sets the *__name__* of the class' type as *name*, which is essentially the Class' Name.

Type str

logger
Shared, class level logger for all instances.

Type logging.Logger

extract()
Extracts Data from a Source and converts it into a PySpark DataFrame.

Returns

Return type pyspark.sql.DataFrame

Note: This method does not take ANY input parameters. All needed parameters are defined in the initialization of the Extractor Object.

Create your own Extractor

Let your extractor class inherit from the extractor base class. This includes the name, string representation and logger attributes from the superclass.

The only mandatory thing is to provide an *extract()* method which
takes
=> *no input parameters*
and **returns** a
=> *PySpark DataFrame!*

All configuration and parameterization should be done while initializing the class instance.

Here would be a simple example for a CSV Extractor:

Exemplary Sample Code

Listing 1: src/spooq2/extractor/csv_extractor.py:

```
from pyspark.sql import SparkSession

from extractor import Extractor

class CSVExtractor(Extractor):
    """
    This is a simplified example on how to implement a new extractor class.
    Please take your time to write proper docstrings as they are automatically
    parsed via Sphinx to build the HTML and PDF documentation.
    Docstrings use the style of Numpy (via the napoleon plug-in).

    This class uses the :meth:`pyspark.sql.DataFrameReader.csv` method internally.

    Examples
    --------
    extracted_df = CSVExtractor(
        input_file='data/input_data.csv'
    ).extract()

    Parameters
    ----------
    input_file: :any:`str`
        The explicit file path for the input data set. Globbing support depends
```

(continues on next page)

(continued from previous page)

```
        on implementation of Spark's csv reader!

    Raises
    ------
    :any:`exceptions.TypeError`:
        path can be only string, list or RDD
    """

    def __init__(self, input_file):
        super(CSVExtractor, self).__init__()
        self.input_file = input_file
        self.spark = SparkSession.Builder()\
            .enableHiveSupport()\
            .appName('spooq2.extractor: {nm}'.format(nm=self.name))\
            .getOrCreate()

    def extract(self):
        self.logger.info('Loading Raw CSV Files from: ' + self.input_file)
        output_df = self.spark.read.load(
            input_file,
            format="csv",
            sep=";",
            inferSchema="true",
            header="true"
        )

        return output_df
```

References to include

Listing 2: src/spooq2/extractor/__init__.py:

```
--- original
+++ adapted
@@ -1,8 +1,10 @@
 from jdbc import JDBCExtractorIncremental, JDBCExtractorFullLoad
 from json_files import JSONExtractor
+from csv_extractor import CSVExtractor

 __all__ = [
     "JDBCExtractorIncremental",
     "JDBCExtractorFullLoad",
     "JSONExtractor",
+    "CSVExtractor",
 ]
```

Tests

One of Spooq2's features is to provide tested code for multiple data pipelines. Please take your time to write sufficient unit tests! You can reuse test data from *tests/data* or create a new schema / data set if needed. A SparkSession is provided as a global fixture called *spark_session*.

Listing 3: tests/unit/extractor/test_csv.py:

```
import pytest

from spooq2.extractor import CSVExtractor

@pytest.fixture()
def default_extractor():
```

(continues on next page)

(continued from previous page)

```
    return CSVExtractor(input_path="data/input_data.csv")


class TestBasicAttributes(object):

    def test_logger_should_be_accessible(self, default_extractor):
        assert hasattr(default_extractor, "logger")

    def test_name_is_set(self, default_extractor):
        assert default_extractor.name == "CSVExtractor"

    def test_str_representation_is_correct(self, default_extractor):
        assert unicode(default_extractor) == "Extractor Object of Class CSVExtractor"

class TestCSVExtraction(object):

    def test_count(default_extractor):
        """Converted DataFrame has the same count as the input data"""
        expected_count = 312
        actual_count = default_extractor.extract().count()
        assert expected_count == actual_count

    def test_schema(default_extractor):
        """Converted DataFrame has the expected schema"""
        do_some_stuff()
        assert expected == actual
```

Documentation

You need to create a *rst* for your extractor which needs to contain at minimum the *automodule* or the *autoclass* directive.

Listing 4: docs/source/extractor/csv.rst:

```
CSV Extractor
=============

Some text if you like...

.. automodule:: spooq2.extractor.csv_extractor
```

To automatically include your new extractor in the HTML documentation you need to add it to a *toctree* directive. Just refer to your newly created *csv.rst* file within the extractor overview page.

Listing 5: docs/source/extractor/overview.rst:

```
--- original
+++ adapted
@@ -7,8 +7,9 @@
 .. toctree::

     json
     jdbc
+    csv

 Class Diagram of Extractor Subpackage
 ---------------------------------------------
 .. uml:: ../diagrams/from_thesis/class_diagram/extractors.puml
    :caption: Class Diagram of Extractor Subpackage
```

That should be all!

1.7.3 Transformer Base Class

Transformers take a `pyspark.sql.DataFrame` as an input, transform it accordingly and return a PySpark DataFrame.

Each Transformer class has to have a *transform* method which takes no arguments and returns a PySpark DataFrame.

Possible transformation methods can be **Selecting the most up to date record by id**, **Exploding an array**, **Filter (on an exploded array)**, **Apply basic threshold cleansing** or **Map the incoming DataFrame to at provided structure**.

class Transformer
Bases: `object`

Base Class of Transformer Classes.

name
Sets the *__name__* of the class' type as *name*, which is essentially the Class' Name.

Type `str`

logger
Shared, class level logger for all instances.

Type `logging.Logger`

transform(*input_df*)
Performs a transformation on a DataFrame.

Parameters **input_df** (`pyspark.sql.DataFrame`) – Input DataFrame

Returns Transformed DataFrame.

Return type `pyspark.sql.DataFrame`

Note: This method does only take the Input DataFrame as a parameters. All other needed parameters are defined in the initialization of the Transformator Object.

Create your own Transformer

Let your transformer class inherit from the transformer base class. This includes the name, string representation and logger attributes from the superclass.

The only mandatory thing is to provide a *transform()* method which
takes a
=> *PySpark DataFrame!*
and **returns** a
=> *PySpark DataFrame!*

All configuration and parameterization should be done while initializing the class instance.

Here would be a simple example for a transformer which drops records without an Id:

Exemplary Sample Code

Listing 6: src/spooq2/transformer/no_id_dropper.py:

```
from transformer import Transformer

class NoIdDropper(Transformer):
    """
```

(continues on next page)

(continued from previous page)

```
    This is a simplified example on how to implement a new transformer class.
    Please take your time to write proper docstrings as they are automatically
    parsed via Sphinx to build the HTML and PDF documentation.
    Docstrings use the style of Numpy (via the napoleon plug-in).

    This class uses the :meth:`pyspark.sql.DataFrame.dropna` method internally.

    Examples
    --------
    input_df = some_extractor_instance.extract()
    transformed_df = NoIdDropper(
        id_columns='user_id'
    ).transform(input_df)

    Parameters
    ----------
    id_columns: :any:`str` or :any:`list`
        The name of the column containing the identifying Id values.
        Defaults to "id"

    Raises
    ------
    :any:`exceptions.ValueError`:
        "how ('" + how + "') should be 'any' or 'all'"
    :any:`exceptions.ValueError`:
        "subset should be a list or tuple of column names"
    """

    def __init__(self, id_columns='id'):
        super(NoIdDropper, self).__init__()
        self.id_columns = id_columns


    def transform(self, input_df):
        self.logger.info("Dropping records without an Id (columns to consider: {col})"
            .format(col=self.id_columns))
        output_df = input_df.dropna(
            how='all',
            thresh=None,
            subset=self.id_columns
        )

        return output_df
```

References to include

This makes it possible to import the new transformer class directly from *spooq2.transformer* instead of *spooq2.transformer.no_id_dropper*. It will also be imported if you use *from spooq2.transformer import **.

Listing 7: src/spooq2/transformer/__init__.py:

```
--- original
+++ adapted
@@ -1,13 +1,15 @@
 from newest_by_group import NewestByGroup
 from mapper import Mapper
 from exploder import Exploder
 from threshold_cleaner import ThresholdCleaner
 from sieve import Sieve
+from no_id_dropper import NoIdDropper

 __all__ = [
```

(continues on next page)

(continued from previous page)

```
      "NewestByGroup",
      "Mapper",
      "Exploder",
      "ThresholdCleaner",
      "Sieve",
+     "NoIdDropper",
 ]
```

Tests

One of Spooq2's features is to provide tested code for multiple data pipelines. Please take your time to write sufficient unit tests! You can reuse test data from *tests/data* or create a new schema / data set if needed. A SparkSession is provided as a global fixture called *spark_session*.

Listing 8: tests/unit/transformer/test_no_id_dropper.py:

```
import pytest
from pyspark.sql.dataframe import DataFrame

from spooq2.transformer import NoIdDropper


@pytest.fixture()
def default_transformer():
    return NoIdDropper(id_columns=["first_name", "last_name"])


@pytest.fixture()
def input_df(spark_session):
    return spark_session.read.parquet("../data/schema_v1/parquetFiles")


@pytest.fixture()
def transformed_df(default_transformer, input_df):
    return default_transformer.transform(input_df)


class TestBasicAttributes(object):

    def test_logger_should_be_accessible(self, default_transformer):
        assert hasattr(default_transformer, "logger")

    def test_name_is_set(self, default_transformer):
        assert default_transformer.name == "NoIdDropper"

    def test_str_representation_is_correct(self, default_transformer):
        assert unicode(default_transformer) == "Transformer Object of Class␣
↪NoIdDropper"

class TestNoIdDropper(object):

    def test_records_are_dropped(transformed_df, input_df):
        """Transformed DataFrame has no records with missing first_name and last_name"
↪""
        assert input_df.where("first_name is null or last_name is null").count() > 0
        assert transformed_df.where("first_name is null or last_name is null").
↪count() == 0

    def test_schema_is_unchanged(transformed_df, input_df):
        """Converted DataFrame has the expected schema"""
        assert transformed_df.schema == input_df.schema
```

Documentation

You need to create a *rst* for your transformer which needs to contain at minimum the *automodule* or the *autoclass* directive.

Listing 9: docs/source/transformer/no_id_dropper.rst:

```
Record Dropper if Id is missing
===============================

Some text if you like...

.. automodule:: spooq2.transformer.no_id_dropper
```

To automatically include your new transformer in the HTML / PDF documentation you need to add it to a *toctree* directive. Just refer to your newly created *no_id_dropper.rst* file within the transformer overview page.

Listing 10: docs/source/transformer/overview.rst:

```
--- original
+++ adapted
@@ -7,14 +7,15 @@
 .. toctree::

     exploder
     sieve
     mapper
     threshold_cleaner
     newest_by_group
+    no_id_dropper

 Class Diagram of Transformer Subpackage
 -----------------------------------------------
 .. uml:: ../diagrams/from_thesis/class_diagram/transformers.puml
    :caption: Class Diagram of Transformer Subpackage
```

That should be it!

1.7.4 Loader Base Class

Loaders take a `pyspark.sql.DataFrame` as an input and save it to a sink.

Each Loader class has to have a *load* method which takes a DataFrame as single paremter.

Possible Loader sinks can be **Hive Tables**, **Kudu Tables**, **HBase Tables**, **JDBC Sinks** or **ParquetFiles**.

class Loader

Bases: `object`

Base Class of Loader Objects.

name

Sets the *__name__* of the class' type as *name*, which is essentially the Class' Name.

Type `str`

logger

Shared, class level logger for all instances.

Type `logging.Logger`

load(*input_df*)

Persists data from a PySpark DataFrame to a target table.

Parameters **input_df** (`pyspark.sql.DataFrame`) – Input DataFrame which has to be loaded to a target destination.

Note: This method takes only a single DataFrame as an input parameter. All other needed parameters are defined in the initialization of the Loader object.

Create your own Loader

Let your loader class inherit from the loader base class. This includes the name, string representation and logger attributes from the superclass.

The only mandatory thing is to provide a *load()* method which
takes a
=> *PySpark DataFrame!*
and **returns**
nothing (or at least the API does not expect anything)

All configuration and parameterization should be done while initializing the class instance.

Here would be a simple example for a loader which save a DataFrame to parquet files:

Exemplary Sample Code

Listing 11: src/spooq2/loader/parquet.py:

```
from pyspark.sql import functions as F

from loader import Loader

class ParquetLoader(loader):
    """
    This is a simplified example on how to implement a new loader class.
    Please take your time to write proper docstrings as they are automatically
    parsed via Sphinx to build the HTML and PDF documentation.
    Docstrings use the style of Numpy (via the napoleon plug-in).

    This class uses the :meth:`pyspark.sql.DataFrameWriter.parquet` method internally.

    Examples
    --------
    input_df = some_extractor_instance.extract()
    output_df = some_transformer_instance.transform(input_df)
    ParquetLoader(
        path="data/parquet_files",
        partition_by="dt",
        explicit_partition_values=20200201,
        compression=""gzip""
    ).load(output_df)

    Parameters
    ----------
    path: :any:`str`
        The path to where the loader persists the output parquet files.
        If partitioning is set, this will be the base path where the partitions
        are stored.

    partition_by: :any:`str` or :any:`list` of (:any:`str`)
        The column name or names by which the output should be partitioned.
        If the partition_by parameter is set to None, no partitioning will be
        performed.
```

(continues on next page)

(continued from previous page)

```
        Defaults to "dt"

    explicit_partition_values: :any:`str` or :any:`int`
                               or :any:`list` of (:any:`str` and :any:`int`)
        Only allowed if partition_by is not None.
        If explicit_partition_values is not None, the dataframe will
            * overwrite the partition_by columns values if it already exists or
            * create and fill the partition_by columns if they do not yet exist
        Defaults to None

    compression: :any:`str`
        The compression codec used for the parquet output files.
        Defaults to "snappy"


    Raises
    ------
    :any:`exceptions.AssertionError`:
        explicit_partition_values can only be used when partition_by is not None
    :any:`exceptions.AssertionError`:
        explicit_partition_values and partition_by must have the same length
    """

    def __init__(self, path, partition_by="dt", explicit_partition_values=None,␣
↪compression_codec="snappy"):
        super(ParquetLoader, self).__init__()
        self.path = path
        self.partition_by = partition_by
        self.explicit_partition_values = explicit_partition_values
        self.compression_codec = compression_codec
        if explicit_partition_values is not None:
            assert (partition_by is not None,
                "explicit_partition_values can only be used when partition_by is not␣
↪None")
            assert (len(partition_by) == len(explicit_partition_values),
                "explicit_partition_values and partition_by must have the same length
↪")

    def load(self, input_df):
        self.logger.info("Persisting DataFrame as Parquet Files to " + self.path)

        if isinstance(self.explicit_partition_values, list):
            for (k, v) in zip(self.partition_by, self.explicit_partition_values):
                input_df = input_df.withColumn(k, F.lit(v))
        elif isinstance(self.explicit_partition_values, basestring):
            input_df = input_df.withColumn(self.partition_by, F.lit(self.explicit_
↪partition_values))

        input_df.write.parquet(
            path=self.path,
            partitionBy=self.partition_by,
            compression=self.compression_codec
        )
```

References to include

This makes it possible to import the new loader class directly from *spooq2.loader* instead of *spooq2.loader.parquet*. It will also be imported if you use *from spooq2.loader import **.

Listing 12: src/spooq2/loader/__init__.py:

```
--- original
+++ adapted
@@ -1,7 +1,9 @@
 from loader import Loader
 from hive_loader import HiveLoader
+from parquet import ParquetLoader

 __all__ = [
     "Loader",
     "HiveLoader",
+    "ParquetLoader",
 ]
```

Tests

One of Spooq2's features is to provide tested code for multiple data pipelines. Please take your time to write sufficient unit tests! You can reuse test data from *tests/data* or create a new schema / data set if needed. A SparkSession is provided as a global fixture called *spark_session*.

Listing 13: tests/unit/loader/test_parquet.py:

```
import pytest
from pyspark.sql.dataframe import DataFrame

from spooq2.loader import ParquetLoader


@pytest.fixture(scope="module")
def output_path(tmpdir_factory):
    return str(tmpdir_factory.mktemp("parquet_output"))


@pytest.fixture(scope="module")
def default_loader(output_path):
    return ParquetLoader(
        path=output_path,
        partition_by="attributes.gender",
        explicit_partition_values=None,
        compression_codec=None
    )


@pytest.fixture(scope="module")
def input_df(spark_session):
    return spark_session.read.parquet("../data/schema_v1/parquetFiles")


@pytest.fixture(scope="module")
def loaded_df(default_loader, input_df, spark_session, output_path):
    default_loader.load(input_df)
    return spark_session.read.parquet(output_path)


class TestBasicAttributes(object):

    def test_logger_should_be_accessible(self, default_loader):
        assert hasattr(default_loader, "logger")

    def test_name_is_set(self, default_loader):
        assert default_loader.name == "ParquetLoader"
```

(continues on next page)

(continued from previous page)

```
    def test_str_representation_is_correct(self, default_loader):
        assert unicode(default_loader) == "loader Object of Class ParquetLoader"

class TestParquetLoader(object):

    def test_count_did_not_change(loaded_df, input_df):
        """Persisted DataFrame has the same number of records than the input DataFrame
↪"""
        assert input_df.count() == output_df.count() and input_df.count() > 0

    def test_schema_is_unchanged(loaded_df, input_df):
        """Loaded DataFrame has the same schema as the input DataFrame"""
        assert loaded.schema == input_df.schema
```

Documentation

You need to create a *rst* for your loader which needs to contain at minimum the *automodule* or the *autoclass* directive.

Listing 14: docs/source/loader/parquet.rst:

```
Parquet Loader
==============================

Some text if you like...

.. automodule:: spooq2.loader.parquet
```

To automatically include your new loader in the HTML / PDF documentation you need to add it to a *toctree* directive. Just refer to your newly created *parquet.rst* file within the loader overview page.

Listing 15: docs/source/loader/overview.rst:

```
--- original
+++ adapted
@@ -7,4 +7,5 @@
 .. toctree::
     hive_loader
+    parquet

 Class Diagram of Loader Subpackage
```

That should be it!

1.8 Setup for Development, Testing, Documenting

Attention: The current version of Spooq is designed (and tested) only for Python 2.7 on ubuntu, manjaro linux and WSL2 (Windows Subsystem Linux).

1.8.1 Prerequisites

- python 2.7
- Java 8 (jdk8-openjdk)
- pipenv
- Latex (for PDF documentation)

1.8.2 Setting up the Environment

The requirements are stored in the file *Pipfile* separated for production and development packages.

To install the packages needed for development and testing run the following command:

```
$ pipenv install --dev
```

This will create a virtual environment in *~/.local/share/virtualenvs*.

If you want to have your virtual environment installed as a sub-folder (.venv) you have to set the environment variable *PIPENV_VENV_IN_PROJECT* to 1.

To remove a virtual environment created with pipenv just change in the folder where you created it and execute *pipenv –rm*.

1.8.3 Activate the Virtual Environment

Listing 16: To activate the virtual environment enter:

```
$ pipenv shell
```

Listing 17: To deactivate the virtual environment simply enter:

```
$ exit
# or close the shell
```

For more commands of pipenv call *pipenv -h*.

1.8.4 Creating Your Own Components

Implementing new extractors, transformers, or loaders is fairly straightforward. Please refer to following descriptions and examples to get an idea:

- *Create your own Extractor*
- *Create your own Transformer*
- *Create your own Loader*

1.8.5 Running Tests

The tests are implemented with the pytest framework.

Listing 18: Start all tests:

```
$ pipenv shell
$ cd tests
$ pytest
```

Test Plugins

Those are the most useful plugins automatically used:

html

Listing 19: Generate an HTML report for the test results:

```
$ pytest --html=report.html
```

random-order

Shuffles the order of execution for the tests to avoid / discover dependencies of the tests.

Randomization is set by a seed number. To re-test the same order of execution where you found an error, just set the seed value to the same as for the failing test. To temporarily disable this feature run with *pytest -p no:random-order -v*

cov

Generates an HTML for the test coverage

Listing 20: Get a test coverage report in the terminal:

```
$ pytest --cov-report term --cov=spooq2
```

Listing 21: Get the test coverage report as HTML

```
$ pytest --cov-report html:cov_html --cov=spooq2
```

ipdb

To use ipdb (IPython Debugger) add following code at your breakpoint::

```
>>> import ipdb
>>> ipdb.set_trace()
```

You have to start pytest with *-s* if you want to use interactive debugger.

```
$ pytest -s
```

1.8.6 Generate Documentation

This project uses Sphinx for creating its documentation. Graphs and diagrams are produced with PlantUML.

The main documentation content is defined as docstrings within the source code. To view the current documentation open *docs/build/html/index.html* or *docs/build/latex/spooq2.pdf* in your application of choice. There are symlinks in the root folder for symplicity:

- Documentation.html
- Documentation.pdf

Although, if you are reading this, you have probably already found the documentation. . .

Diagrams

For generating the graphs and diagrams, you need a working plantuml installation on your computer! Please refer to sphinxcontrib-plantuml.

HTML

```
$ cd docs
$ make html
$ chromium build/html/index.html
```

PDF

For generating documentation in the PDF format you need to have a working (pdf)latex installation on your computer! Please refer to TexLive on how to install TeX Live - a compatible latex distribution. But beware, the download size is huge!

```
$ cd docs
$ make latexpdf
$ evince build/latex/Spooq2.pdf
```

Configuration

Themes, plugins, settings, ... are defined in *docs/source/conf.py*.

napoleon

Enables support for parsing docstrings in NumPy / Google Style

intersphinx

Allows linking to other projects' documentation. E.g., PySpark, Python2 To add an external project, at the documentation link to *intersphinx_mapping* in *conf.py*

recommonmark

This allows you to write CommonMark (Markdown) inside of Docutils & Sphinx projects instead of rst.

plantuml

Allows for inline Plant UML code (uml directive) which is automatically rendered into an svg image and placed in the document. Allows also to source puml-files. See *Architecture Overview* for an example.

1.9 Architecture Overview

1.9.1 Typical Data Flow of a Spooq Data Pipeline

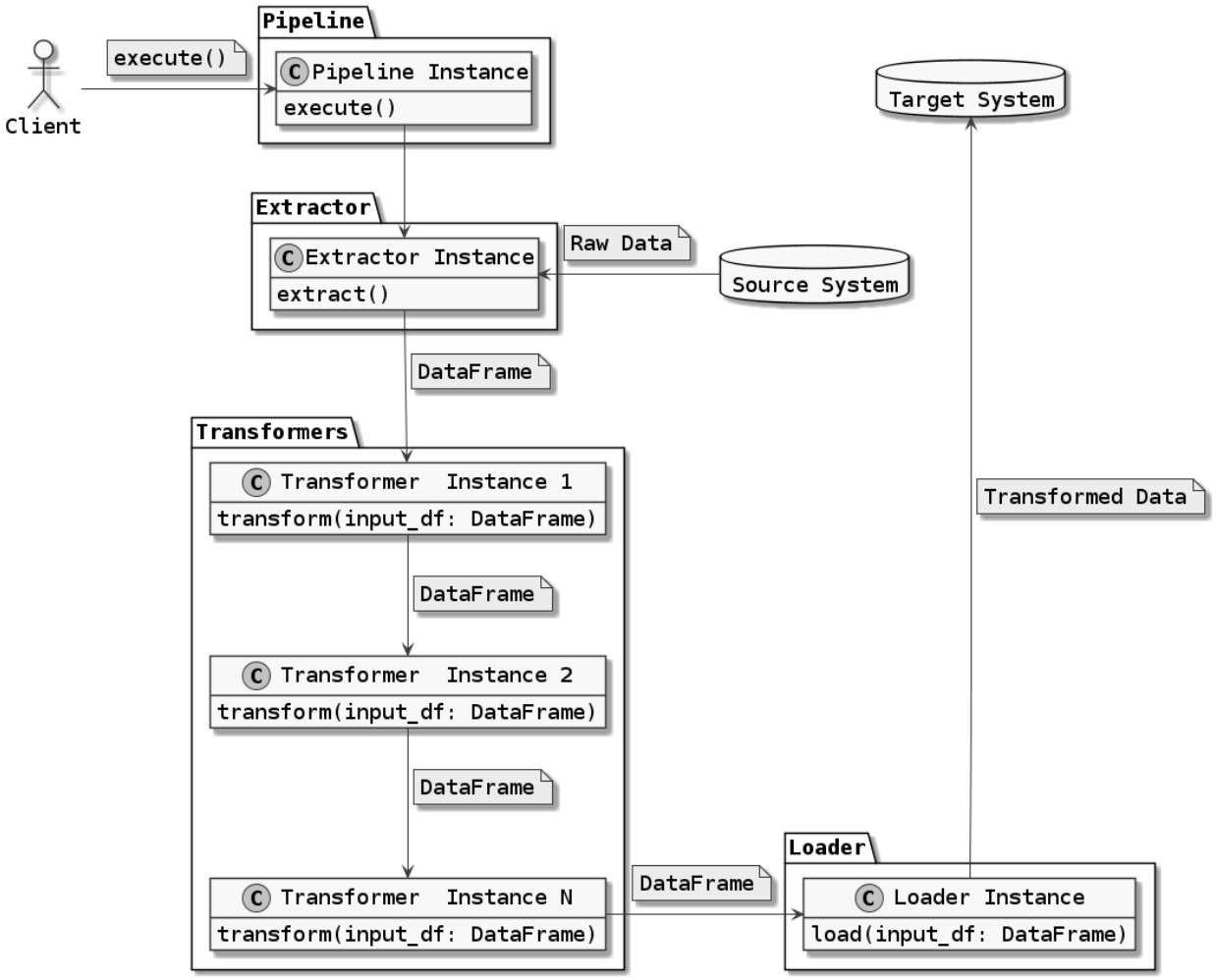


Fig. 7: Typical Data Flow of a Spooq Data Pipeline

1.9.2 Simplified Class Diagram

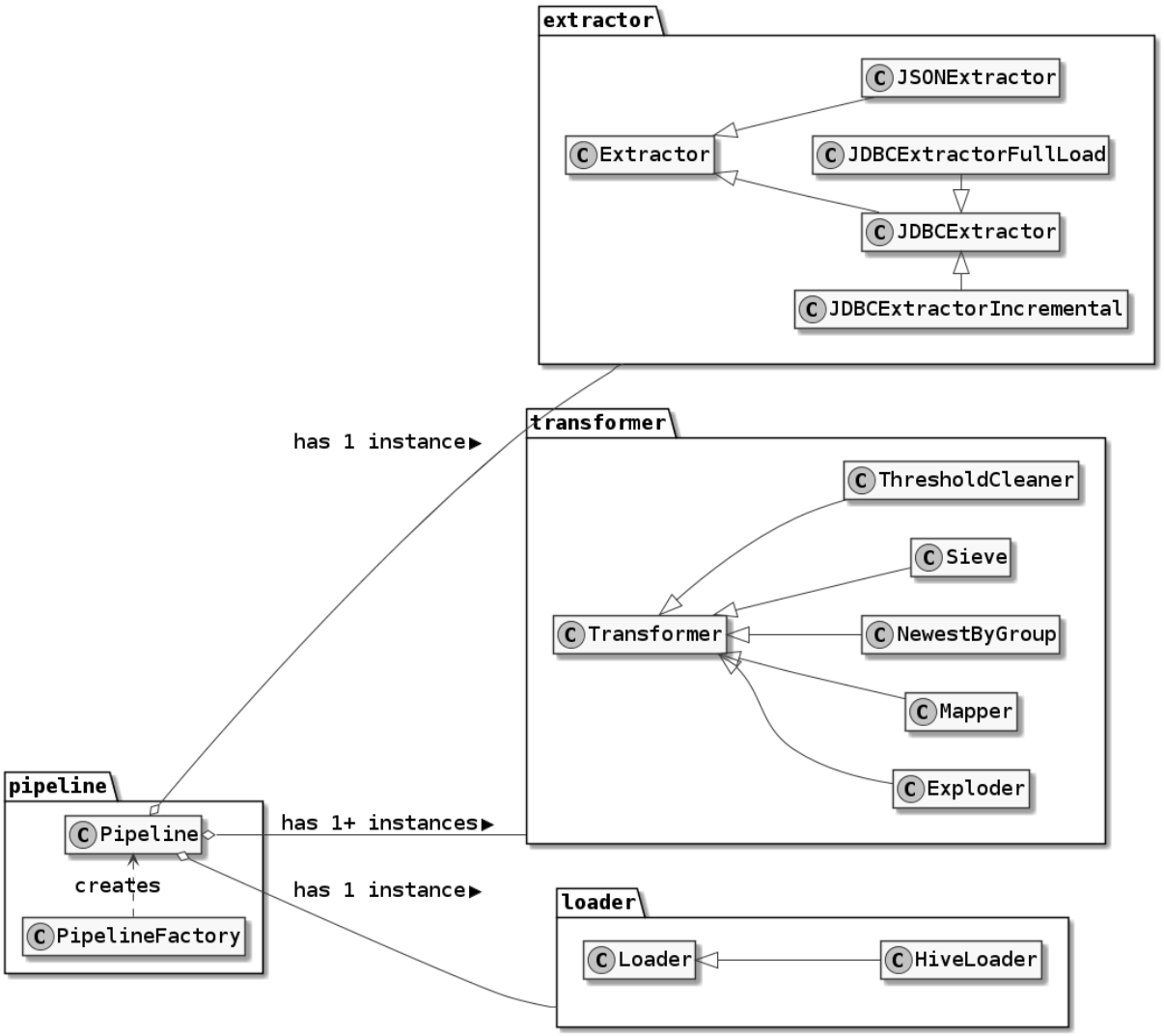


Fig. 8: Simplified Class Diagram

CHAPTER 2

Indices and tables

- modindex
- search

Python Module Index

S

Index

Symbols

A

C

E

G

H

I

J

L

M

N

P

S

T

U

Appendix B: Preparation of Yelp's Raw Data for Examples

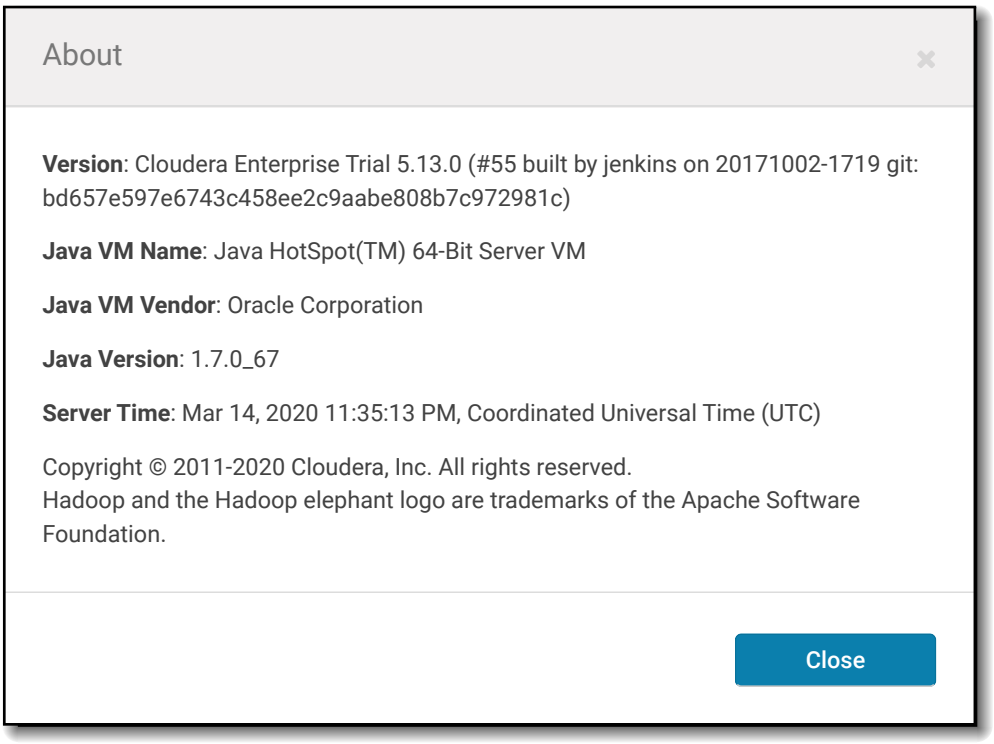
Preprocessing of Yelp Dataset

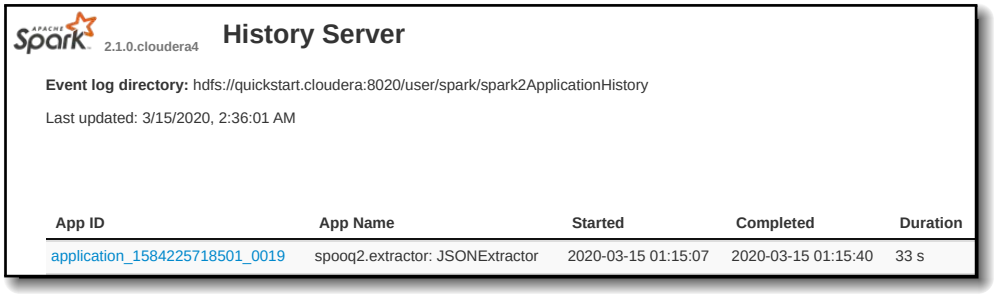
```
import pyspark.sql.functions as F
import pyspark.sql.types as sql_types
from random import SystemRandom

"""helper udf"""
@F.udf("string")
def rand_year(x):
    rnd = SystemRandom()
    return str(rnd.randint(2018, 2018)).rjust(4, "0")

@F.udf("string")
def rand_month(x):
    rnd = SystemRandom()
    return str(rnd.randint(9, 11)).rjust(2, "0")

@F.udf("string")
def rand_day(x):
    rnd = SystemRandom()
    return str(rnd.randint(1, 29)).rjust(2, "0")


def remove_empty_elements_in_array_func(x):
    elements = []
    for elem in x:
        if elem:
            elements.append(elem)
    return elements

remove_empty_elements_in_array = F.udf(
    remove_empty_elements_in_array_func,
    sql_types.ArrayType(sql_types.StringType()))


"""user dataset"""
df = spark.read.json("source/user.json")
df = df.withColumn("friends", F.split(df.friends, ", "))
```

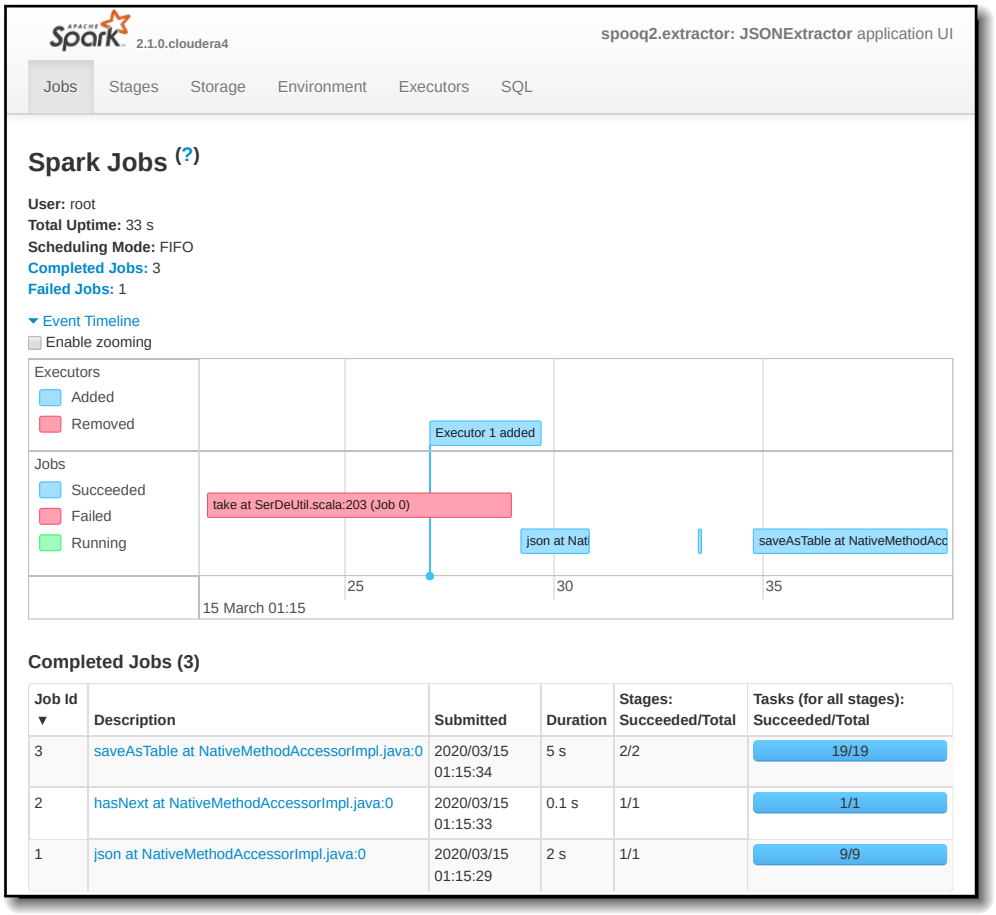
```
df = df.withColumn("elite", F.split(df.elite, ","))
df = df.withColumn("elite", remove_empty_elements_in_array(df.elite))
df = df.withColumn("p_year", F.substring(df.yelping_since, 1, 4))
df = df.withColumn("p_month", F.substring(df.yelping_since, 6, 2))
df = df.withColumn("p_day", F.substring(df.yelping_since, 9, 2))

df.write.partitionBy(
    "p_year", "p_month", "p_day"
).json("user", compression="gzip")

"""business dataset"""
df = spark.read.json("source/business.json")
df = df.withColumn("categories", F.split(df.categories, ", "))
df = df.withColumn("p_year", rand_year(df.business_id))
df = df.withColumn("p_month", rand_month(df.business_id))
df = df.withColumn("p_day", rand_day(df.business_id))

df.write.partitionBy(
    "p_year", "p_month", "p_day"
).json("business", compression="gzip")

"""review dataset"""
df = spark.read.json("source/review.json")
df = df.withColumn("p_year", F.substring(df.date, 1, 4))
df = df.withColumn("p_month", F.substring(df.date, 6, 2))
df = df.withColumn("p_day", F.substring(df.date, 9, 2))

df.write.partitionBy(
    "p_year", "p_month", "p_day"
).json("review", compression="gzip")

"""check_in dataset"""
df = spark.read.json("source/checkin.json")
df = df.withColumn("p_year", rand_year(df.business_id))
df = df.withColumn("p_month", rand_month(df.business_id))
df = df.withColumn("p_day", rand_day(df.business_id))

df.write.partitionBy(
    "p_year", "p_month", "p_day"
).json("checkin", compression="gzip")

"""check_in dataset"""
df = spark.read.json("source/tip.json")
df = df.withColumn("p_year", F.substring(df.date, 1, 4))
df = df.withColumn("p_month", F.substring(df.date, 6, 2))
df = df.withColumn("p_day", F.substring(df.date, 9, 2))

df.write.partitionBy(
    "p_year", "p_month", "p_day"
).json("tip", compression="gzip")
```

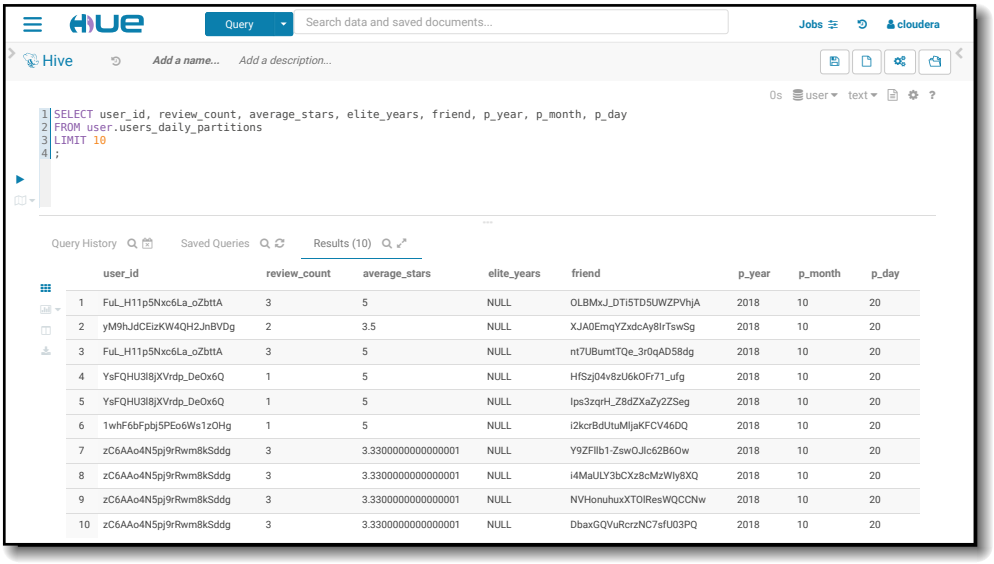
Appendix C: Demonstration in Different Environments

Spark on Hadoop Distribution (Cloudera)

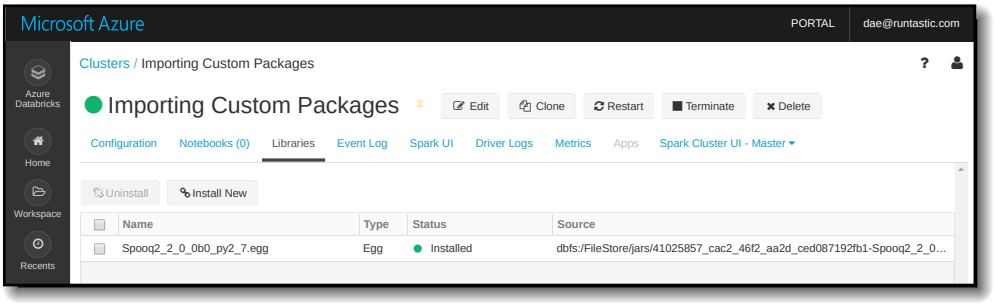


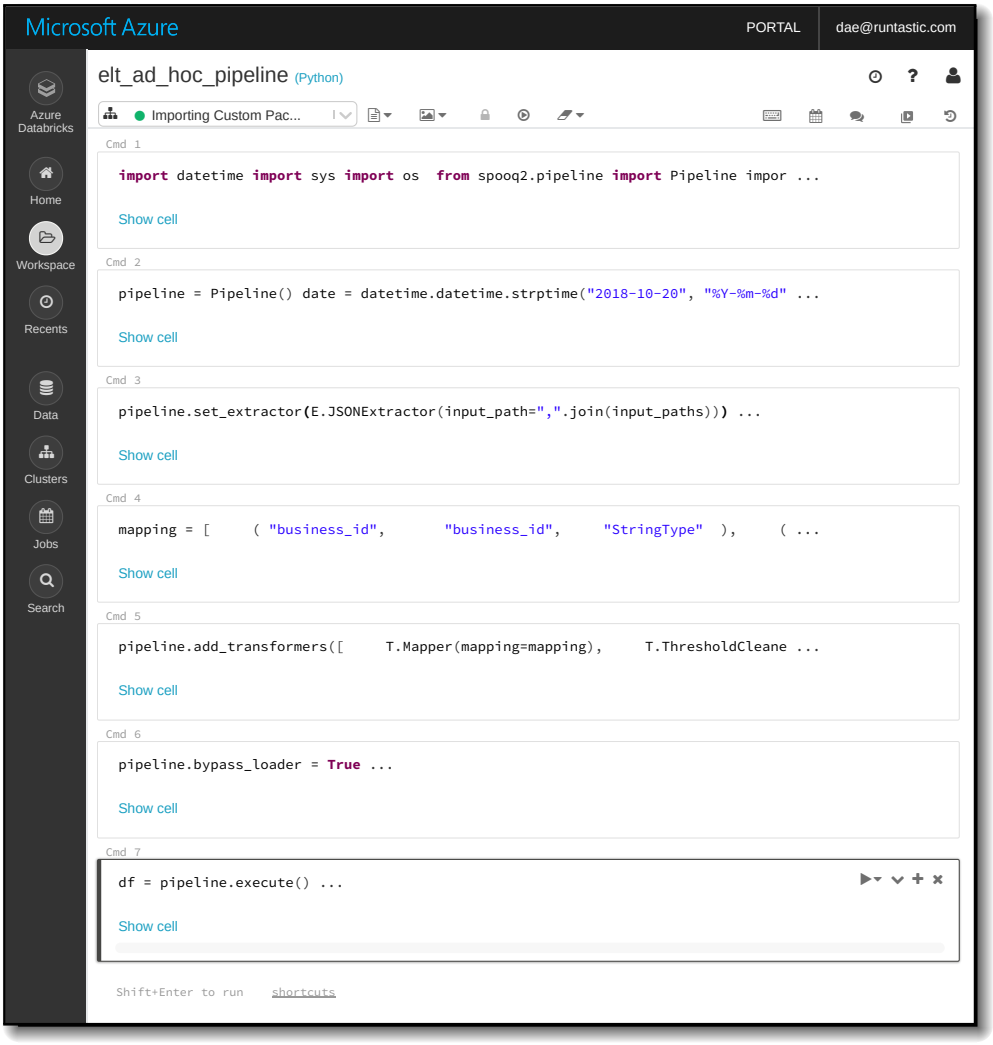

2.1.0.cloudera4
History Server
Event log directory: hdfs://quickstart.cloudera:8020/user/spark/spark2ApplicationHistory
Last updated: 3/15/2020, 2:36:01 AM
App ID
App Name
Started
Completed
Duration
application_1584225718501_0019
spooq2.extractor: JSONExtractor
2020-03-15 01:15:07
2020-03-15 01:15:40
33 s


2.1.0.cloudera4
spooq2.extractor: JSONExtractor application UI
Jobs
Stages
Storage
Environment
Executors
SQL
Spark Jobs (?)
User: root
Total Uptime: 33 s
Scheduling Mode: FIFO
Completed Jobs: 3
Failed Jobs: 1
Event Timeline
Enable zooming
Executors
Added
Removed
Executor 1 added
Jobs
Succeeded
Failed
Running
take at SerDeUtil.scala:203 (Job 0)
json at Nati
saveAsTable at NativeMethodAcc
25
30
35
15 March 01:15
Completed Jobs (3)
Job Id ▼
Description
Submitted
Duration
Stages: Succeeded/Total
Tasks (for all stages): Succeeded/Total
3
saveAsTable at NativeMethodAccessorImpl.java:0
2020/03/15 01:15:34
5 s
2/2
19/19
2
hasNext at NativeMethodAccessorImpl.java:0
2020/03/15 01:15:33
0.1 s
1/1
1/1
1
json at NativeMethodAccessorImpl.java:0
2020/03/15 01:15:29
2 s
1/1
9/9



Spark Cloud Distribution (Databricks)






```
1  import datetime
2  import sys
3  import os
4
5  from spooq2.pipeline import Pipeline
6  import spooq2.extractor as   E
7  import spooq2.transformer as T
8  import spooq2.loader as      L
```

Command took 0.06 seconds -- by dae@runtastic.com at 3/21/2020, 2:27:01 AM on Importing Custom Packages

```
pipeline = Pipeline()
date = datetime.datetime.strptime("2018-10-20", "%Y-%m-%d")

input_paths = []
for delta in range(0,7):
    day = date - datetime.timedelta(delta)
    partition_path = datetime.datetime.strftime(
        day, "p_year=%Y/p_month=%m/p_day=%d"
    )
    input_paths.append(os.path.join("/user/dae@runtastic.com/yelp_data_set", partition_path))
```

Command took 0.05 seconds -- by dae@runtastic.com at 3/21/2020, 2:27:25 AM on Importing Custom Packages

```
pipeline.set_extractor(E.JSONExtractor(input_path=",".join(input_paths)))
```

Command took 0.06 seconds -- by dae@runtastic.com at 3/21/2020, 2:27:47 AM on Importing Custom Packages

```
mapping = [
    ( "business_id",       "business_id",     "StringType"  ),
    ( "name",              "name",            "StringType"  ),
    ( "address",           "address",         "StringType"  ),
    ( "city",              "city",            "StringType"  ),
    ( "state",             "state",           "StringType"  ),
    ( "postal_code",       "postal_code",     "StringType"  ),
    ( "latitude",          "latitude",        "DoubleType"  ),
    ( "longitude",         "longitude",       "DoubleType"  ),
    ( "stars",             "stars",           "LongType"    ),
    ( "review_count",      "review_count",    "LongType"    ),
    ( "categories",        "categories",      "json_string" ),
    ( "open_on_monday",    "hours.Monday",    "StringType"  ),
    ( "open_on_tuesday",   "hours.Tuesday",   "StringType"  ),
    ( "open_on_wednesday", "hours.Wednesday", "StringType"  ),
    ( "open_on_thursday",  "hours.Thursday",  "StringType"  ),
    ( "open_on_friday",    "hours.Friday",    "StringType"  ),
    ( "open_on_saturday",  "hours.Saturday",  "StringType"  ),
    ( "attributes",        "attributes",      "json_string" ),
]

```

Command took 0.09 seconds -- by dae@runtastic.com at 3/21/2020, 2:28:01 AM on Importing Custom Packages

```
pipeline.add_transformers([
    T.Mapper(mapping=mapping),
    T.ThresholdCleaner(thresholds={
        "stars": {"min": 1, "max": 5},
        "latitude": {"min": -90.0, "max": 90.0},
        "longitude": {"min": -180.0, "max": 180.0}
    }),
])
```

```
pipeline.bypass_loader = True
```

```
Command took 0.04 seconds -- by dae@runtastic.com at 3/21/2020, 2:28:07 AM on Importing Custom Packages
```

```
df = pipeline.execute()
```

```
(2) Spark Jobs
  Job 2  View (Stages: 0/0, 1 skipped)
    Stage 2: 0/1 skipped
  Job 3  View (Stages: 1/1)
    Stage 3: 28/28
df: pyspark.sql.dataframe.DataFrame
    business_id: string
    name: string
    address: string
    city: string
    state: string
    postal_code: string
    latitude: double
    longitude: double
    stars: long
    review_count: long
    categories: string
    open_on_monday: string
    open_on_tuesday: string
    open_on_wednesday: string
    open_on_thursday: string
    open_on_friday: string
    open_on_saturday: string
    attributes: string
```

```
Command took 3.47 seconds -- by dae@runtastic.com at 3/21/2020, 2:28:12 AM on Importing Custom Packages
```

Appendix D: Demonstration of Semi-Automatic Configuration by Reasoning

Rules Triggered by ETL Batch Pipeline Inference

spooq_rules Logs

```
DEBUG:experta.watchers.AGENDA:0: 'set_time_range_for_last_day' '<f-1>, <f-0>'
DEBUG:experta.watchers.AGENDA:1: 'set_pipeline_type_according_to_set_batch_size' '<f-1>, <f-0>'
INFO:experta.watchers.RULES:FIRE 1 set_pipeline_type_according_to_set_batch_size: <f-1>, <f-0>
INFO:experta.watchers.FACTS: ==> <f-2>: Fact(pipeline_type='batch')
INFO:experta.watchers.ACTIVATIONS: <== 'set_pipeline_type_according_to_set_batch_size': <f-0>, <f-1> [EXECUTED]
INFO:experta.watchers.ACTIVATIONS: ==> 'set_level_of_detail_for_batch': <f-0>, <f-2>
DEBUG:experta.watchers.AGENDA:0: 'set_time_range_for_last_day' '<f-1>, <f-0>'
DEBUG:experta.watchers.AGENDA:1: 'set_level_of_detail_for_batch' '<f-0>, <f-2>'
INFO:experta.watchers.RULES:FIRE 2 set_level_of_detail_for_batch: <f-0>, <f-2>
INFO:experta.watchers.FACTS: ==> <f-3>: Fact(level_of_detail='std')
INFO:experta.watchers.ACTIVATIONS: <== 'set_level_of_detail_for_batch': <f-0>, <f-2> [EXECUTED]
INFO:experta.watchers.ACTIVATIONS: ==> 'set_integer_to_level_of_detail': <f-3>
DEBUG:experta.watchers.AGENDA:0: 'set_time_range_for_last_day' '<f-1>, <f-0>'
DEBUG:experta.watchers.AGENDA:1: 'set_integer_to_level_of_detail' '<f-3>'
INFO:experta.watchers.RULES:FIRE 3 set_integer_to_level_of_detail: <f-3>
INFO:experta.watchers.FACTS: ==> <f-4>: Fact(level_of_detail_int=5)
```

```
DEBUG:experta.watchers.AGENDA:0: 'set_time_range_for_last_day' '<f-1>, <f-0>'
INFO:experta.watchers.RULES:FIRE 4 set_time_range_for_last_day: <f-1>, <f-0>
INFO:experta.watchers.FACTS: ==> <f-5>: Fact(time_range='last_day')
INFO:experta.watchers.ACTIVATIONS: <== 'set_time_range_for_last_day': <f-0>, <f-1> [EXECUTED]
INFO:werkzeug:127.0.0.1 - - [18/Mar/2020 00:16:01] "POST /context/get HTTP/1.1" 200 -
DEBUG:experta.watchers.AGENDA:0: 'json_extractor' '<f-1>'
INFO:experta.watchers.RULES:FIRE 1 json_extractor: <f-1>
INFO:werkzeug:127.0.0.1 - - [18/Mar/2020 00:16:01] "POST /extractor/name HTTP/1.1" 200 -
DEBUG:experta.watchers.AGENDA:0: 'input_from_yesterday' '<f-1>'
INFO:experta.watchers.RULES:FIRE 1 input_from_yesterday: <f-1>
INFO:experta.watchers.FACTS: ==> <f-2>: Fact(input=<frozendict {'path': 'user/p_year=2018/p_month=10/p_day=20'}>)
INFO:experta.watchers.ACTIVATIONS: ==> 'return_result': <f-2>
DEBUG:experta.watchers.AGENDA:0: 'return_result' '<f-2>'
INFO:experta.watchers.RULES:FIRE 2 return_result: <f-2>
INFO:werkzeug:127.0.0.1 - - [18/Mar/2020 00:16:01] "POST /extractor/params/JSONExtractor HTTP/1.1" 200 -
DEBUG:experta.watchers.AGENDA:0: 'mapping_provided' '<f-1>'
DEBUG:experta.watchers.AGENDA:1: 'arrays_to_explode_defined' '<f-1>'
DEBUG:experta.watchers.AGENDA:2: 'filter_expressions_provided' '<f-1>'
DEBUG:experta.watchers.AGENDA:3: 'needs_cleansing' '<f-1>'
DEBUG:experta.watchers.AGENDA:4: 'needs_deduplication' '<f-1>'
INFO:experta.watchers.RULES:FIRE 1 needs_deduplication: <f-1>
DEBUG:experta.watchers.AGENDA:0: 'mapping_provided' '<f-1>'
DEBUG:experta.watchers.AGENDA:1: 'arrays_to_explode_defined' '<f-1>'
DEBUG:experta.watchers.AGENDA:2: 'filter_expressions_provided' '<f-1>'
DEBUG:experta.watchers.AGENDA:3: 'needs_cleansing' '<f-1>'
INFO:experta.watchers.RULES:FIRE 2 needs_cleansing: <f-1>
DEBUG:experta.watchers.AGENDA:0: 'mapping_provided' '<f-1>'
DEBUG:experta.watchers.AGENDA:1: 'arrays_to_explode_defined' '<f-1>'
DEBUG:experta.watchers.AGENDA:2: 'filter_expressions_provided' '<f-1>'
INFO:experta.watchers.RULES:FIRE 3 filter_expressions_provided: <f-1>
DEBUG:experta.watchers.AGENDA:0: 'mapping_provided' '<f-1>'
DEBUG:experta.watchers.AGENDA:1: 'arrays_to_explode_defined' '<f-1>'
INFO:experta.watchers.RULES:FIRE 4 arrays_to_explode_defined: <f-1>
DEBUG:experta.watchers.AGENDA:0: 'mapping_provided' '<f-1>'
INFO:experta.watchers.RULES:FIRE 5 mapping_provided: <f-1>
INFO:werkzeug:127.0.0.1 - - [18/Mar/2020 00:16:01] "POST /transformer/names HTTP/1.1" 200 -
DEBUG:experta.watchers.AGENDA:0: 'return_result' '<f-1>'
INFO:experta.watchers.RULES:FIRE 1 return_result: <f-1>
INFO:werkzeug:127.0.0.1 - - [18/Mar/2020 00:16:01] "POST /transformer/params/Exploder HTTP/1.1" 200 -
DEBUG:experta.watchers.AGENDA:0: 'return_result' '<f-1>'
INFO:experta.watchers.RULES:FIRE 1 return_result: <f-1>
INFO:werkzeug:127.0.0.1 - - [18/Mar/2020 00:16:01] "POST /transformer/params/Sieve HTTP/1.1" 200 -
DEBUG:experta.watchers.AGENDA:0: 'return_result' '<f-1>'
INFO:experta.watchers.RULES:FIRE 1 return_result: <f-1>
INFO:werkzeug:127.0.0.1 - - [18/Mar/2020 00:16:01] "POST /transformer/params/Sieve HTTP/1.1" 200 -
DEBUG:experta.watchers.AGENDA:0: 'reason_over_each_column_and_return_results' '<f-1>'
INFO:experta.watchers.RULES:FIRE 1 reason_over_each_column_and_return_results: <f-1>
```

```
INFO:experta.watchers.FACTS: ==> <f-0>: InitialFact()
INFO:experta.watchers.ACTIVATIONS: ==> 'set_target_type_to_default': <f-0>
INFO:experta.watchers.ACTIVATIONS: ==> 'set_pii_to_default': <f-0>
INFO:experta.watchers.FACTS: ==> <f-1>: Fact(name='user_id',
↪  target_type='StringType', triviality=1, desc='22 character unique user
↪  id, maps to the user in user.json')
INFO:experta.watchers.ACTIVATIONS: <== 'set_target_type_to_default': <f-0>
↪  [EXECUTED]
INFO:experta.watchers.ACTIVATIONS: ==> 'set_name_as_path': <f-0>, <f-1>
DEBUG:experta.watchers.AGENDA:0: 'set_pii_to_default' '<f-0>'
DEBUG:experta.watchers.AGENDA:1: 'set_name_as_path' '<f-0>, <f-1>'
INFO:experta.watchers.RULES:FIRE 1 set_name_as_path: <f-0>, <f-1>
INFO:experta.watchers.FACTS: ==> <f-2>: Fact(path='user_id')
INFO:experta.watchers.ACTIVATIONS: <== 'set_name_as_path': <f-0>, <f-1>
↪  [EXECUTED]
DEBUG:experta.watchers.AGENDA:0: 'set_pii_to_default' '<f-0>'
INFO:experta.watchers.RULES:FIRE 2 set_pii_to_default: <f-0>
INFO:experta.watchers.FACTS: ==> <f-3>: Fact(has_pii='no')
INFO:experta.watchers.ACTIVATIONS: <== 'set_pii_to_default': <f-0>
↪  [EXECUTED]
INFO:experta.watchers.ACTIVATIONS: ==> 'return_result': <f-2>, <f-3>,
↪  <f-1>
DEBUG:experta.watchers.AGENDA:0: 'return_result' '<f-2>, <f-3>, <f-1>'
INFO:experta.watchers.RULES:FIRE 3 return_result: <f-2>, <f-3>, <f-1>
INFO:experta.watchers.FACTS: ==> <f-0>: InitialFact()
INFO:experta.watchers.ACTIVATIONS: ==> 'set_target_type_to_default': <f-0>
INFO:experta.watchers.ACTIVATIONS: ==> 'set_pii_to_default': <f-0>
INFO:experta.watchers.FACTS: ==> <f-1>: Fact(name='review_count',
↪  target_type='LongType', triviality=1, desc="the number of reviews
↪  they've written")
INFO:experta.watchers.ACTIVATIONS: <== 'set_target_type_to_default': <f-0>
↪  [EXECUTED]
INFO:experta.watchers.ACTIVATIONS: ==> 'set_name_as_path': <f-0>, <f-1>
DEBUG:experta.watchers.AGENDA:0: 'set_pii_to_default' '<f-0>'
DEBUG:experta.watchers.AGENDA:1: 'set_name_as_path' '<f-0>, <f-1>'
INFO:experta.watchers.RULES:FIRE 1 set_name_as_path: <f-0>, <f-1>
INFO:experta.watchers.FACTS: ==> <f-2>: Fact(path='review_count')
INFO:experta.watchers.ACTIVATIONS: <== 'set_name_as_path': <f-0>, <f-1>
↪  [EXECUTED]
DEBUG:experta.watchers.AGENDA:0: 'set_pii_to_default' '<f-0>'
INFO:experta.watchers.RULES:FIRE 2 set_pii_to_default: <f-0>
INFO:experta.watchers.FACTS: ==> <f-3>: Fact(has_pii='no')
INFO:experta.watchers.ACTIVATIONS: <== 'set_pii_to_default': <f-0>
↪  [EXECUTED]
INFO:experta.watchers.ACTIVATIONS: ==> 'return_result': <f-3>, <f-1>,
↪  <f-2>
DEBUG:experta.watchers.AGENDA:0: 'return_result' '<f-3>, <f-1>, <f-2>'
INFO:experta.watchers.RULES:FIRE 3 return_result: <f-3>, <f-1>, <f-2>
INFO:experta.watchers.FACTS: ==> <f-0>: InitialFact()
INFO:experta.watchers.ACTIVATIONS: ==> 'set_target_type_to_default': <f-0>
INFO:experta.watchers.ACTIVATIONS: ==> 'set_pii_to_default': <f-0>
INFO:experta.watchers.FACTS: ==> <f-1>: Fact(name='average_stars',
↪  target_type='DoubleType', triviality=1, desc='average rating of all
↪  reviews')
INFO:experta.watchers.ACTIVATIONS: <== 'set_target_type_to_default': <f-0>
↪  [EXECUTED]
INFO:experta.watchers.ACTIVATIONS: ==> 'set_name_as_path': <f-0>, <f-1>
DEBUG:experta.watchers.AGENDA:0: 'set_pii_to_default' '<f-0>'
DEBUG:experta.watchers.AGENDA:1: 'set_name_as_path' '<f-0>, <f-1>'
INFO:experta.watchers.RULES:FIRE 1 set_name_as_path: <f-0>, <f-1>
```

```
INFO:experta.watchers.FACTS: ==> <f-2>: Fact(path='average_stars')
INFO:experta.watchers.ACTIVATIONS: <== 'set_name_as_path': <f-0>, <f-1>
↪  [EXECUTED]
DEBUG:experta.watchers.AGENDA:0: 'set_pii_to_default' '<f-0>'
INFO:experta.watchers.RULES:FIRE 2 set_pii_to_default: <f-0>
INFO:experta.watchers.FACTS: ==> <f-3>: Fact(has_pii='no')
INFO:experta.watchers.ACTIVATIONS: <== 'set_pii_to_default': <f-0>
↪  [EXECUTED]
INFO:experta.watchers.ACTIVATIONS: ==> 'return_result': <f-3>, <f-2>,
↪  <f-1>
DEBUG:experta.watchers.AGENDA:0: 'return_result' '<f-3>, <f-2>, <f-1>'
INFO:experta.watchers.RULES:FIRE 3 return_result: <f-3>, <f-2>, <f-1>
INFO:experta.watchers.FACTS: ==> <f-0>: InitialFact()
INFO:experta.watchers.ACTIVATIONS: ==> 'set_target_type_to_default': <f-0>
INFO:experta.watchers.ACTIVATIONS: ==> 'set_pii_to_default': <f-0>
INFO:experta.watchers.FACTS: ==> <f-1>: Fact(name='elite_years',
↪  path='elite', target_type='json_string', triviality=5, desc='the years
↪  the user was elite')
INFO:experta.watchers.ACTIVATIONS: <== 'set_target_type_to_default': <f-0>
↪  [EXECUTED]
DEBUG:experta.watchers.AGENDA:0: 'set_pii_to_default' '<f-0>'
INFO:experta.watchers.RULES:FIRE 1 set_pii_to_default: <f-0>
INFO:experta.watchers.FACTS: ==> <f-2>: Fact(has_pii='no')
INFO:experta.watchers.ACTIVATIONS: <== 'set_pii_to_default': <f-0>
↪  [EXECUTED]
INFO:experta.watchers.ACTIVATIONS: ==> 'return_result': <f-1>, <f-2>
DEBUG:experta.watchers.AGENDA:0: 'return_result' '<f-1>, <f-2>'
INFO:experta.watchers.RULES:FIRE 2 return_result: <f-1>, <f-2>
INFO:experta.watchers.FACTS: ==> <f-0>: InitialFact()
INFO:experta.watchers.ACTIVATIONS: ==> 'set_target_type_to_default': <f-0>
INFO:experta.watchers.ACTIVATIONS: ==> 'set_pii_to_default': <f-0>
INFO:experta.watchers.FACTS: ==> <f-1>: Fact(name='friend',
↪  path='friends_element', target_type='StringType', triviality=1,
↪  desc="the user's friend as user_ids")
INFO:experta.watchers.ACTIVATIONS: <== 'set_target_type_to_default': <f-0>
↪  [EXECUTED]
DEBUG:experta.watchers.AGENDA:0: 'set_pii_to_default' '<f-0>'
INFO:experta.watchers.RULES:FIRE 1 set_pii_to_default: <f-0>
INFO:experta.watchers.FACTS: ==> <f-2>: Fact(has_pii='no')
INFO:experta.watchers.ACTIVATIONS: <== 'set_pii_to_default': <f-0>
↪  [EXECUTED]
INFO:experta.watchers.ACTIVATIONS: ==> 'return_result': <f-1>, <f-2>
DEBUG:experta.watchers.AGENDA:0: 'return_result' '<f-1>, <f-2>'
INFO:experta.watchers.RULES:FIRE 2 return_result: <f-1>, <f-2>
INFO:werkzeug:127.0.0.1 - - [18/Mar/2020 00:16:01] "POST
↪  /transformer/params/Mapper HTTP/1.1" 200 -
DEBUG:experta.watchers.AGENDA:0: 'return_result' '<f-1>'
INFO:experta.watchers.RULES:FIRE 1 return_result: <f-1>
INFO:werkzeug:127.0.0.1 - - [18/Mar/2020 00:16:01] "POST
↪  /transformer/params/ThresholdCleaner HTTP/1.1" 200 -
DEBUG:experta.watchers.AGENDA:0: 'return_result' '<f-1>'
INFO:experta.watchers.RULES:FIRE 1 return_result: <f-1>
INFO:werkzeug:127.0.0.1 - - [18/Mar/2020 00:16:01] "POST
↪  /transformer/params/NewestByGroup HTTP/1.1" 200 -
DEBUG:experta.watchers.AGENDA:0: 'hive_extractor' '<f-1>'
INFO:experta.watchers.RULES:FIRE 1 hive_extractor: <f-1>
INFO:werkzeug:127.0.0.1 - - [18/Mar/2020 00:16:01] "POST /loader/name
↪  HTTP/1.1" 200 -
DEBUG:experta.watchers.AGENDA:0: 'set_default_for_clear_partition' '<f-0>'
```

```
DEBUG:experta.watchers.AGENDA:1:
↪  'set_default_for_overwrite_partition_value' '<f-0>'
DEBUG:experta.watchers.AGENDA:2: 'set_default_for_auto_create_table'
↪  '<f-0>'
DEBUG:experta.watchers.AGENDA:3: 'set_entity_name_as_db_name_name' '<f-1>,
↪  <f-0>'
DEBUG:experta.watchers.AGENDA:4: 'fix_values_in_partition_definitions'
↪  '<f-1>'
INFO:experta.watchers.RULES:FIRE 1 fix_values_in_partition_definitions:
↪  <f-1>
INFO:experta.watchers.FACTS: ==> <f-0>: InitialFact()
INFO:experta.watchers.ACTIVATIONS: ==> 'set_default_for_data_type': <f-0>
INFO:experta.watchers.FACTS: ==> <f-1>: Fact(column_name='p_year',
↪  date='2018-10-20')
INFO:experta.watchers.ACTIVATIONS: ==> 'set_default_value_to_year': <f-1>,
↪  <f-0>
INFO:experta.watchers.ACTIVATIONS: ==> 'set_default_value_to_none': <f-1>,
↪  <f-0>
DEBUG:experta.watchers.AGENDA:0: 'set_default_for_data_type' '<f-0>'
DEBUG:experta.watchers.AGENDA:1: 'set_default_value_to_none' '<f-1>,
↪  <f-0>'
DEBUG:experta.watchers.AGENDA:2: 'set_default_value_to_year' '<f-1>,
↪  <f-0>'
INFO:experta.watchers.RULES:FIRE 1 set_default_value_to_year: <f-1>, <f-0>
INFO:experta.watchers.FACTS: ==> <f-2>: Fact(default_value=2018)
INFO:experta.watchers.ACTIVATIONS: <== 'set_default_value_to_year': <f-0>,
↪  <f-1> [EXECUTED]
INFO:experta.watchers.ACTIVATIONS: <== 'set_default_value_to_none': <f-0>,
↪  <f-1> [EXECUTED]
DEBUG:experta.watchers.AGENDA:0: 'set_default_for_data_type' '<f-0>'
INFO:experta.watchers.RULES:FIRE 2 set_default_for_data_type: <f-0>
INFO:experta.watchers.FACTS: ==> <f-3>: Fact(column_type='IntegerType')
INFO:experta.watchers.ACTIVATIONS: <== 'set_default_for_data_type': <f-0>
↪  [EXECUTED]
INFO:experta.watchers.ACTIVATIONS: ==> 'return_result': <f-2>, <f-1>,
↪  <f-3>
DEBUG:experta.watchers.AGENDA:0: 'return_result' '<f-2>, <f-1>, <f-3>'
INFO:experta.watchers.RULES:FIRE 3 return_result: <f-2>, <f-1>, <f-3>
INFO:experta.watchers.FACTS: ==> <f-0>: InitialFact()
INFO:experta.watchers.ACTIVATIONS: ==> 'set_default_for_data_type': <f-0>
INFO:experta.watchers.FACTS: ==> <f-1>: Fact(column_name='p_month',
↪  date='2018-10-20')
INFO:experta.watchers.ACTIVATIONS: ==> 'set_default_value_to_month':
↪  <f-0>, <f-1>
INFO:experta.watchers.ACTIVATIONS: ==> 'set_default_value_to_none': <f-0>,
↪  <f-1>
DEBUG:experta.watchers.AGENDA:0: 'set_default_for_data_type' '<f-0>'
DEBUG:experta.watchers.AGENDA:1: 'set_default_value_to_none' '<f-0>,
↪  <f-1>'
DEBUG:experta.watchers.AGENDA:2: 'set_default_value_to_month' '<f-0>,
↪  <f-1>'
INFO:experta.watchers.RULES:FIRE 1 set_default_value_to_month: <f-0>,
↪  <f-1>
INFO:experta.watchers.FACTS: ==> <f-2>: Fact(default_value=10)
INFO:experta.watchers.ACTIVATIONS: <== 'set_default_value_to_month':
↪  <f-0>, <f-1> [EXECUTED]
INFO:experta.watchers.ACTIVATIONS: <== 'set_default_value_to_none': <f-0>,
↪  <f-1> [EXECUTED]
DEBUG:experta.watchers.AGENDA:0: 'set_default_for_data_type' '<f-0>'
INFO:experta.watchers.RULES:FIRE 2 set_default_for_data_type: <f-0>
```

```
INFO:experta.watchers.FACTS: ==> <f-3>: Fact(column_type='IntegerType')
INFO:experta.watchers.ACTIVATIONS: <== 'set_default_for_data_type': <f-0>
    [EXECUTED]
INFO:experta.watchers.ACTIVATIONS: ==> 'return_result': <f-2>, <f-1>,
    <f-3>
DEBUG:experta.watchers.AGENDA:0: 'return_result' '<f-2>, <f-1>, <f-3>'
INFO:experta.watchers.RULES:FIRE 3 return_result: <f-2>, <f-1>, <f-3>
INFO:experta.watchers.FACTS: ==> <f-0>: InitialFact()
INFO:experta.watchers.ACTIVATIONS: ==> 'set_default_for_data_type': <f-0>
INFO:experta.watchers.FACTS: ==> <f-1>: Fact(column_name='p_day',
    date='2018-10-20')
INFO:experta.watchers.ACTIVATIONS: ==> 'set_default_value_to_day': <f-0>,
    <f-1>
INFO:experta.watchers.ACTIVATIONS: ==> 'set_default_value_to_none': <f-0>,
    <f-1>
DEBUG:experta.watchers.AGENDA:0: 'set_default_for_data_type' '<f-0>'
DEBUG:experta.watchers.AGENDA:1: 'set_default_value_to_none' '<f-0>,
    <f-1>'
DEBUG:experta.watchers.AGENDA:2: 'set_default_value_to_day' '<f-0>, <f-1>'
INFO:experta.watchers.RULES:FIRE 1 set_default_value_to_day: <f-0>, <f-1>
INFO:experta.watchers.FACTS: ==> <f-2>: Fact(default_value=20)
INFO:experta.watchers.ACTIVATIONS: <== 'set_default_value_to_day': <f-0>,
    <f-1> [EXECUTED]
INFO:experta.watchers.ACTIVATIONS: <== 'set_default_value_to_none': <f-0>,
    <f-1> [EXECUTED]
DEBUG:experta.watchers.AGENDA:0: 'set_default_for_data_type' '<f-0>'
INFO:experta.watchers.RULES:FIRE 2 set_default_for_data_type: <f-0>
INFO:experta.watchers.FACTS: ==> <f-3>: Fact(column_type='IntegerType')
INFO:experta.watchers.ACTIVATIONS: <== 'set_default_for_data_type': <f-0>
    [EXECUTED]
INFO:experta.watchers.ACTIVATIONS: ==> 'return_result': <f-2>, <f-1>,
    <f-3>
DEBUG:experta.watchers.AGENDA:0: 'return_result' '<f-2>, <f-1>, <f-3>'
INFO:experta.watchers.RULES:FIRE 3 return_result: <f-2>, <f-1>, <f-3>
DEBUG:experta.watchers.AGENDA:0: 'set_default_for_clear_partition' '<f-0>'
DEBUG:experta.watchers.AGENDA:1:
    'set_default_for_overwrite_partition_value' '<f-0>'
DEBUG:experta.watchers.AGENDA:2: 'set_default_for_auto_create_table'
    '<f-0>'
DEBUG:experta.watchers.AGENDA:3: 'set_entity_name_as_db_name_name' '<f-1>,
    <f-0>'
INFO:experta.watchers.RULES:FIRE 2 set_entity_name_as_db_name_name: <f-1>,
    <f-0>
INFO:experta.watchers.FACTS: ==> <f-2>: Fact(output=<frozendict
    {'db_name': 'user'}>)
INFO:experta.watchers.ACTIVATIONS: <== 'set_entity_name_as_db_name_name':
    <f-1>, <f-0> [EXECUTED]
INFO:experta.watchers.ACTIVATIONS: ==>
    'set_table_prefix_from_db_name_name': <f-2>, <f-0>
DEBUG:experta.watchers.AGENDA:0: 'set_default_for_clear_partition' '<f-0>'
DEBUG:experta.watchers.AGENDA:1:
    'set_default_for_overwrite_partition_value' '<f-0>'
DEBUG:experta.watchers.AGENDA:2: 'set_default_for_auto_create_table'
    '<f-0>'
DEBUG:experta.watchers.AGENDA:3: 'set_table_prefix_from_db_name_name'
    '<f-2>, <f-0>'
INFO:experta.watchers.RULES:FIRE 3 set_table_prefix_from_db_name_name:
    <f-2>, <f-0>
INFO:experta.watchers.FACTS: ==> <f-3>: Fact(output=<frozendict
    {'table_prefix': 'user'}>)
```

```
INFO:experta.watchers.ACTIVATIONS: <==
↪  'set_table_prefix_from_db_name_name': <f-2>, <f-0> [EXECUTED]
INFO:experta.watchers.ACTIVATIONS: ==> 'set_table_name_with_prefix':
↪  <f-1>, <f-3>, <f-0>
DEBUG:experta.watchers.AGENDA:0: 'set_default_for_clear_partition' '<f-0>'
DEBUG:experta.watchers.AGENDA:1:
↪  'set_default_for_overwrite_partition_value' '<f-0>'
DEBUG:experta.watchers.AGENDA:2: 'set_default_for_auto_create_table'
↪  '<f-0>'
DEBUG:experta.watchers.AGENDA:3: 'set_table_name_with_prefix' '<f-1>,
↪  <f-3>, <f-0>'
INFO:experta.watchers.RULES:FIRE 4 set_table_name_with_prefix: <f-1>,
↪  <f-3>, <f-0>
INFO:experta.watchers.FACTS: ==> <f-4>: Fact(output=<frozendict
↪  {'table_name': 'users_daily_partitions'}>)
INFO:experta.watchers.ACTIVATIONS: <== 'set_table_name_with_prefix':
↪  <f-1>, <f-0>, <f-3> [EXECUTED]
DEBUG:experta.watchers.AGENDA:0: 'set_default_for_clear_partition' '<f-0>'
DEBUG:experta.watchers.AGENDA:1:
↪  'set_default_for_overwrite_partition_value' '<f-0>'
DEBUG:experta.watchers.AGENDA:2: 'set_default_for_auto_create_table'
↪  '<f-0>'
INFO:experta.watchers.RULES:FIRE 5 set_default_for_auto_create_table:
↪  <f-0>
INFO:experta.watchers.FACTS: ==> <f-5>: Fact(output=<frozendict
↪  {'auto_create_table': True}>)
INFO:experta.watchers.ACTIVATIONS: <==
↪  'set_default_for_auto_create_table': <f-0> [EXECUTED]
DEBUG:experta.watchers.AGENDA:0: 'set_default_for_clear_partition' '<f-0>'
DEBUG:experta.watchers.AGENDA:1:
↪  'set_default_for_overwrite_partition_value' '<f-0>'
INFO:experta.watchers.RULES:FIRE 6
↪  set_default_for_overwrite_partition_value: <f-0>
INFO:experta.watchers.FACTS: ==> <f-6>: Fact(output=<frozendict
↪  {'overwrite_partition_value': True}>)
INFO:experta.watchers.ACTIVATIONS: <==
↪  'set_default_for_overwrite_partition_value': <f-0> [EXECUTED]
DEBUG:experta.watchers.AGENDA:0: 'set_default_for_clear_partition' '<f-0>'
INFO:experta.watchers.RULES:FIRE 7 set_default_for_clear_partition: <f-0>
INFO:experta.watchers.FACTS: ==> <f-7>: Fact(output=<frozendict
↪  {'clear_partition': True}>)
INFO:experta.watchers.ACTIVATIONS: <== 'set_default_for_clear_partition':
↪  <f-0> [EXECUTED]
INFO:experta.watchers.ACTIVATIONS: ==> 'return_result': <f-5>, <f-2>,
↪  <f-1>, <f-7>, <f-4>, <f-6>
DEBUG:experta.watchers.AGENDA:0: 'return_result' '<f-5>, <f-2>, <f-1>,
↪  <f-7>, <f-4>, <f-6>'
INFO:experta.watchers.RULES:FIRE 8 return_result: <f-5>, <f-2>, <f-1>,
↪  <f-7>, <f-4>, <f-6>
INFO:werkzeug:127.0.0.1 - - [18/Mar/2020 00:16:01] "POST
↪  /loader/params/HiveLoader HTTP/1.1" 200 -
INFO:werkzeug:127.0.0.1 - - [18/Mar/2020 00:16:01] "POST /pipeline/get
↪  HTTP/1.1" 200 -
```

Rules Triggered by ELT Ad Hoc Pipeline Inference

spooq_rules Logs

```
DEBUG:experta.watchers.AGENDA:0: 'set_default_pipeline_type' '<f-0>'
INFO:experta.watchers.RULES:FIRE 1 set_default_pipeline_type: <f-0>
INFO:experta.watchers.FACTS: ==> <f-2>: Fact(pipeline_type='ad_hoc',
↪  batch_size='no')
INFO:experta.watchers.ACTIVATIONS: <== 'set_default_pipeline_type': <f-0>
↪  [EXECUTED]
INFO:experta.watchers.ACTIVATIONS: ==> 'set_level_of_detail_for_ad_hoc':
↪  <f-2>, <f-0>
DEBUG:experta.watchers.AGENDA:0: 'set_level_of_detail_for_ad_hoc' '<f-2>,
↪  <f-0>'
INFO:experta.watchers.RULES:FIRE 2 set_level_of_detail_for_ad_hoc: <f-2>,
↪  <f-0>
INFO:experta.watchers.FACTS: ==> <f-3>: Fact(level_of_detail='all')
INFO:experta.watchers.ACTIVATIONS: <== 'set_level_of_detail_for_ad_hoc':
↪  <f-2>, <f-0> [EXECUTED]
INFO:experta.watchers.ACTIVATIONS: ==> 'set_integer_to_level_of_detail':
↪  <f-3>
DEBUG:experta.watchers.AGENDA:0: 'set_integer_to_level_of_detail' '<f-3>'
INFO:experta.watchers.RULES:FIRE 3 set_integer_to_level_of_detail: <f-3>
INFO:experta.watchers.FACTS: ==> <f-4>: Fact(level_of_detail_int=10)
INFO:werkzeug:127.0.0.1 - - [18/Mar/2020 00:45:34] "POST /context/get
↪  HTTP/1.1" 200 -
DEBUG:experta.watchers.AGENDA:0: 'json_extractor' '<f-1>'
INFO:experta.watchers.RULES:FIRE 1 json_extractor: <f-1>
INFO:werkzeug:127.0.0.1 - - [18/Mar/2020 00:45:34] "POST /extractor/name
↪  HTTP/1.1" 200 -
DEBUG:experta.watchers.AGENDA:0: 'input_from_last_week' '<f-1>'
INFO:experta.watchers.RULES:FIRE 1 input_from_last_week: <f-1>
INFO:experta.watchers.FACTS: ==> <f-2>: Fact(input=<frozendict {'path':
↪  'business/p_year=2018/p_month=10/p_day=20,
↪  business/p_year=2018/p_month=10/p_day=19,
↪  business/p_year=2018/p_month=1
INFO:experta.watchers.ACTIVATIONS: ==> 'return_result': <f-2>
DEBUG:experta.watchers.AGENDA:0: 'return_result' '<f-2>'
INFO:experta.watchers.RULES:FIRE 2 return_result: <f-2>
INFO:werkzeug:127.0.0.1 - - [18/Mar/2020 00:45:34] "POST
↪  /extractor/params/JSONExtractor HTTP/1.1" 200 -
DEBUG:experta.watchers.AGENDA:0: 'mapping_provided' '<f-1>'
DEBUG:experta.watchers.AGENDA:1: 'needs_cleansing' '<f-1>'
INFO:experta.watchers.RULES:FIRE 1 needs_cleansing: <f-1>
DEBUG:experta.watchers.AGENDA:0: 'mapping_provided' '<f-1>'
INFO:experta.watchers.RULES:FIRE 2 mapping_provided: <f-1>
INFO:werkzeug:127.0.0.1 - - [18/Mar/2020 00:45:34] "POST
↪  /transformer/names HTTP/1.1" 200 -
DEBUG:experta.watchers.AGENDA:0:
↪  'reason_over_each_column_and_return_results' '<f-1>'
INFO:experta.watchers.RULES:FIRE 1
↪  reason_over_each_column_and_return_results: <f-1>
INFO:experta.watchers.FACTS: ==> <f-0>: InitialFact()
INFO:experta.watchers.ACTIVATIONS: ==> 'set_target_type_to_default': <f-0>
INFO:experta.watchers.ACTIVATIONS: ==> 'set_pii_to_default': <f-0>
```

```
INFO:experta.watchers.FACTS: ==> <f-1>: Fact(name='business_id',
↪  triviality=1, desc='22 character unique string')
INFO:experta.watchers.ACTIVATIONS: ==> 'set_name_as_path': <f-0>, <f-1>
DEBUG:experta.watchers.AGENDA:0: 'set_target_type_to_default' '<f-0>'
DEBUG:experta.watchers.AGENDA:1: 'set_pii_to_default' '<f-0>'
DEBUG:experta.watchers.AGENDA:2: 'set_name_as_path' '<f-0>, <f-1>'
INFO:experta.watchers.RULES:FIRE 1 set_name_as_path: <f-0>, <f-1>
INFO:experta.watchers.FACTS: ==> <f-2>: Fact(path='business_id')
INFO:experta.watchers.ACTIVATIONS: <== 'set_name_as_path': <f-0>, <f-1>
↪  [EXECUTED]
DEBUG:experta.watchers.AGENDA:0: 'set_target_type_to_default' '<f-0>'
DEBUG:experta.watchers.AGENDA:1: 'set_pii_to_default' '<f-0>'
INFO:experta.watchers.RULES:FIRE 2 set_pii_to_default: <f-0>
INFO:experta.watchers.FACTS: ==> <f-3>: Fact(has_pii='no')
INFO:experta.watchers.ACTIVATIONS: <== 'set_pii_to_default': <f-0>
↪  [EXECUTED]
DEBUG:experta.watchers.AGENDA:0: 'set_target_type_to_default' '<f-0>'
INFO:experta.watchers.RULES:FIRE 3 set_target_type_to_default: <f-0>
INFO:experta.watchers.FACTS: ==> <f-4>: Fact(target_type='StringType')
INFO:experta.watchers.ACTIVATIONS: <== 'set_target_type_to_default': <f-0>
↪  [EXECUTED]
INFO:experta.watchers.ACTIVATIONS: ==> 'return_result': <f-3>, <f-4>,
↪  <f-2>, <f-1>
DEBUG:experta.watchers.AGENDA:0: 'return_result' '<f-3>, <f-4>, <f-2>,
↪  <f-1>'
INFO:experta.watchers.RULES:FIRE 4 return_result: <f-3>, <f-4>, <f-2>,
↪  <f-1>
INFO:experta.watchers.FACTS: ==> <f-0>: InitialFact()
INFO:experta.watchers.ACTIVATIONS: ==> 'set_target_type_to_default': <f-0>
INFO:experta.watchers.ACTIVATIONS: ==> 'set_pii_to_default': <f-0>
INFO:experta.watchers.FACTS: ==> <f-1>: Fact(name='name', triviality=1)
INFO:experta.watchers.ACTIVATIONS: ==> 'set_name_as_path': <f-1>, <f-0>
DEBUG:experta.watchers.AGENDA:0: 'set_target_type_to_default' '<f-0>'
DEBUG:experta.watchers.AGENDA:1: 'set_pii_to_default' '<f-0>'
DEBUG:experta.watchers.AGENDA:2: 'set_name_as_path' '<f-1>, <f-0>'
INFO:experta.watchers.RULES:FIRE 1 set_name_as_path: <f-1>, <f-0>
INFO:experta.watchers.FACTS: ==> <f-2>: Fact(path='name')
INFO:experta.watchers.ACTIVATIONS: <== 'set_name_as_path': <f-1>, <f-0>
↪  [EXECUTED]
DEBUG:experta.watchers.AGENDA:0: 'set_target_type_to_default' '<f-0>'
DEBUG:experta.watchers.AGENDA:1: 'set_pii_to_default' '<f-0>'
INFO:experta.watchers.RULES:FIRE 2 set_pii_to_default: <f-0>
INFO:experta.watchers.FACTS: ==> <f-3>: Fact(has_pii='no')
INFO:experta.watchers.ACTIVATIONS: <== 'set_pii_to_default': <f-0>
↪  [EXECUTED]
DEBUG:experta.watchers.AGENDA:0: 'set_target_type_to_default' '<f-0>'
INFO:experta.watchers.RULES:FIRE 3 set_target_type_to_default: <f-0>
INFO:experta.watchers.FACTS: ==> <f-4>: Fact(target_type='StringType')
INFO:experta.watchers.ACTIVATIONS: <== 'set_target_type_to_default': <f-0>
↪  [EXECUTED]
INFO:experta.watchers.ACTIVATIONS: ==> 'return_result': <f-3>, <f-1>,
↪  <f-4>, <f-2>
DEBUG:experta.watchers.AGENDA:0: 'return_result' '<f-3>, <f-1>, <f-4>,
↪  <f-2>'
INFO:experta.watchers.RULES:FIRE 4 return_result: <f-3>, <f-1>, <f-4>,
↪  <f-2>
INFO:experta.watchers.FACTS: ==> <f-0>: InitialFact()
INFO:experta.watchers.ACTIVATIONS: ==> 'set_target_type_to_default': <f-0>
INFO:experta.watchers.ACTIVATIONS: ==> 'set_pii_to_default': <f-0>
```

```
INFO:experta.watchers.FACTS: ==> <f-1>: Fact(name='address',
    ↪  triviality=10)
INFO:experta.watchers.ACTIVATIONS: ==> 'set_name_as_path': <f-0>, <f-1>
DEBUG:experta.watchers.AGENDA:0: 'set_target_type_to_default' '<f-0>'
DEBUG:experta.watchers.AGENDA:1: 'set_pii_to_default' '<f-0>'
DEBUG:experta.watchers.AGENDA:2: 'set_name_as_path' '<f-0>, <f-1>'
INFO:experta.watchers.RULES:FIRE 1 set_name_as_path: <f-0>, <f-1>
INFO:experta.watchers.FACTS: ==> <f-2>: Fact(path='address')
INFO:experta.watchers.ACTIVATIONS: <== 'set_name_as_path': <f-0>, <f-1>
    ↪  [EXECUTED]
DEBUG:experta.watchers.AGENDA:0: 'set_target_type_to_default' '<f-0>'
DEBUG:experta.watchers.AGENDA:1: 'set_pii_to_default' '<f-0>'
INFO:experta.watchers.RULES:FIRE 2 set_pii_to_default: <f-0>
INFO:experta.watchers.FACTS: ==> <f-3>: Fact(has_pii='no')
INFO:experta.watchers.ACTIVATIONS: <== 'set_pii_to_default': <f-0>
    ↪  [EXECUTED]
DEBUG:experta.watchers.AGENDA:0: 'set_target_type_to_default' '<f-0>'
INFO:experta.watchers.RULES:FIRE 3 set_target_type_to_default: <f-0>
INFO:experta.watchers.FACTS: ==> <f-4>: Fact(target_type='StringType')
INFO:experta.watchers.ACTIVATIONS: <== 'set_target_type_to_default': <f-0>
    ↪  [EXECUTED]
INFO:experta.watchers.ACTIVATIONS: ==> 'return_result': <f-3>, <f-2>,
    ↪  <f-4>, <f-1>
DEBUG:experta.watchers.AGENDA:0: 'return_result' '<f-3>, <f-2>, <f-4>,
    ↪  <f-1>'
INFO:experta.watchers.RULES:FIRE 4 return_result: <f-3>, <f-2>, <f-4>,
    ↪  <f-1>
INFO:experta.watchers.FACTS: ==> <f-0>: InitialFact()
INFO:experta.watchers.ACTIVATIONS: ==> 'set_target_type_to_default': <f-0>
INFO:experta.watchers.ACTIVATIONS: ==> 'set_pii_to_default': <f-0>
INFO:experta.watchers.FACTS: ==> <f-1>: Fact(name='city')
INFO:experta.watchers.ACTIVATIONS: ==> 'set_name_as_path': <f-1>, <f-0>
DEBUG:experta.watchers.AGENDA:0: 'set_target_type_to_default' '<f-0>'
DEBUG:experta.watchers.AGENDA:1: 'set_pii_to_default' '<f-0>'
DEBUG:experta.watchers.AGENDA:2: 'set_name_as_path' '<f-1>, <f-0>'
INFO:experta.watchers.RULES:FIRE 1 set_name_as_path: <f-1>, <f-0>
INFO:experta.watchers.FACTS: ==> <f-2>: Fact(path='city')
INFO:experta.watchers.ACTIVATIONS: <== 'set_name_as_path': <f-0>, <f-1>
    ↪  [EXECUTED]
DEBUG:experta.watchers.AGENDA:0: 'set_target_type_to_default' '<f-0>'
DEBUG:experta.watchers.AGENDA:1: 'set_pii_to_default' '<f-0>'
INFO:experta.watchers.RULES:FIRE 2 set_pii_to_default: <f-0>
INFO:experta.watchers.FACTS: ==> <f-3>: Fact(has_pii='no')
INFO:experta.watchers.ACTIVATIONS: <== 'set_pii_to_default': <f-0>
    ↪  [EXECUTED]
DEBUG:experta.watchers.AGENDA:0: 'set_target_type_to_default' '<f-0>'
INFO:experta.watchers.RULES:FIRE 3 set_target_type_to_default: <f-0>
INFO:experta.watchers.FACTS: ==> <f-4>: Fact(target_type='StringType')
INFO:experta.watchers.ACTIVATIONS: <== 'set_target_type_to_default': <f-0>
    ↪  [EXECUTED]
INFO:experta.watchers.ACTIVATIONS: ==> 'return_result': <f-3>, <f-1>,
    ↪  <f-4>, <f-2>
DEBUG:experta.watchers.AGENDA:0: 'return_result' '<f-3>, <f-1>, <f-4>,
    ↪  <f-2>'
INFO:experta.watchers.RULES:FIRE 4 return_result: <f-3>, <f-1>, <f-4>,
    ↪  <f-2>
INFO:experta.watchers.FACTS: ==> <f-0>: InitialFact()
INFO:experta.watchers.ACTIVATIONS: ==> 'set_target_type_to_default': <f-0>
INFO:experta.watchers.ACTIVATIONS: ==> 'set_pii_to_default': <f-0>
INFO:experta.watchers.FACTS: ==> <f-1>: Fact(name='state')
```

```
INFO:experta.watchers.ACTIVATIONS: ==> 'set_name_as_path': <f-0>, <f-1>
DEBUG:experta.watchers.AGENDA:0: 'set_target_type_to_default' '<f-0>'
DEBUG:experta.watchers.AGENDA:1: 'set_pii_to_default' '<f-0>'
DEBUG:experta.watchers.AGENDA:2: 'set_name_as_path' '<f-0>, <f-1>'
INFO:experta.watchers.RULES:FIRE 1 set_name_as_path: <f-0>, <f-1>
INFO:experta.watchers.FACTS: ==> <f-2>: Fact(path='state')
INFO:experta.watchers.ACTIVATIONS: <== 'set_name_as_path': <f-0>, <f-1>
↪  [EXECUTED]
DEBUG:experta.watchers.AGENDA:0: 'set_target_type_to_default' '<f-0>'
DEBUG:experta.watchers.AGENDA:1: 'set_pii_to_default' '<f-0>'
INFO:experta.watchers.RULES:FIRE 2 set_pii_to_default: <f-0>
INFO:experta.watchers.FACTS: ==> <f-3>: Fact(has_pii='no')
INFO:experta.watchers.ACTIVATIONS: <== 'set_pii_to_default': <f-0>
↪  [EXECUTED]
DEBUG:experta.watchers.AGENDA:0: 'set_target_type_to_default' '<f-0>'
INFO:experta.watchers.RULES:FIRE 3 set_target_type_to_default: <f-0>
INFO:experta.watchers.FACTS: ==> <f-4>: Fact(target_type='StringType')
INFO:experta.watchers.ACTIVATIONS: <== 'set_target_type_to_default': <f-0>
↪  [EXECUTED]
INFO:experta.watchers.ACTIVATIONS: ==> 'return_result': <f-1>, <f-3>,
↪  <f-4>, <f-2>
DEBUG:experta.watchers.AGENDA:0: 'return_result' '<f-1>, <f-3>, <f-4>,
↪  <f-2>'
INFO:experta.watchers.RULES:FIRE 4 return_result: <f-1>, <f-3>, <f-4>,
↪  <f-2>
INFO:experta.watchers.FACTS: ==> <f-0>: InitialFact()
INFO:experta.watchers.ACTIVATIONS: ==> 'set_target_type_to_default': <f-0>
INFO:experta.watchers.ACTIVATIONS: ==> 'set_pii_to_default': <f-0>
INFO:experta.watchers.FACTS: ==> <f-1>: Fact(name='postal_code')
INFO:experta.watchers.ACTIVATIONS: ==> 'set_name_as_path': <f-1>, <f-0>
DEBUG:experta.watchers.AGENDA:0: 'set_target_type_to_default' '<f-0>'
DEBUG:experta.watchers.AGENDA:1: 'set_pii_to_default' '<f-0>'
DEBUG:experta.watchers.AGENDA:2: 'set_name_as_path' '<f-1>, <f-0>'
INFO:experta.watchers.RULES:FIRE 1 set_name_as_path: <f-1>, <f-0>
INFO:experta.watchers.FACTS: ==> <f-2>: Fact(path='postal_code')
INFO:experta.watchers.ACTIVATIONS: <== 'set_name_as_path': <f-1>, <f-0>
↪  [EXECUTED]
DEBUG:experta.watchers.AGENDA:0: 'set_target_type_to_default' '<f-0>'
DEBUG:experta.watchers.AGENDA:1: 'set_pii_to_default' '<f-0>'
INFO:experta.watchers.RULES:FIRE 2 set_pii_to_default: <f-0>
INFO:experta.watchers.FACTS: ==> <f-3>: Fact(has_pii='no')
INFO:experta.watchers.ACTIVATIONS: <== 'set_pii_to_default': <f-0>
↪  [EXECUTED]
DEBUG:experta.watchers.AGENDA:0: 'set_target_type_to_default' '<f-0>'
INFO:experta.watchers.RULES:FIRE 3 set_target_type_to_default: <f-0>
INFO:experta.watchers.FACTS: ==> <f-4>: Fact(target_type='StringType')
INFO:experta.watchers.ACTIVATIONS: <== 'set_target_type_to_default': <f-0>
↪  [EXECUTED]
INFO:experta.watchers.ACTIVATIONS: ==> 'return_result': <f-2>, <f-1>,
↪  <f-3>, <f-4>
DEBUG:experta.watchers.AGENDA:0: 'return_result' '<f-2>, <f-1>, <f-3>,
↪  <f-4>'
INFO:experta.watchers.RULES:FIRE 4 return_result: <f-2>, <f-1>, <f-3>,
↪  <f-4>
INFO:experta.watchers.FACTS: ==> <f-0>: InitialFact()
INFO:experta.watchers.ACTIVATIONS: ==> 'set_target_type_to_default': <f-0>
INFO:experta.watchers.ACTIVATIONS: ==> 'set_pii_to_default': <f-0>
INFO:experta.watchers.FACTS: ==> <f-1>: Fact(name='latitude',
↪  target_type='DoubleType')
```

```
INFO:experta.watchers.ACTIVATIONS: <== 'set_target_type_to_default': <f-0> [EXECUTED]
INFO:experta.watchers.ACTIVATIONS: ==> 'set_name_as_path': <f-1>, <f-0>
DEBUG:experta.watchers.AGENDA:0: 'set_pii_to_default' '<f-0>'
DEBUG:experta.watchers.AGENDA:1: 'set_name_as_path' '<f-1>, <f-0>'
INFO:experta.watchers.RULES:FIRE 1 set_name_as_path: <f-1>, <f-0>
INFO:experta.watchers.FACTS: ==> <f-2>: Fact(path='latitude')
INFO:experta.watchers.ACTIVATIONS: <== 'set_name_as_path': <f-1>, <f-0> [EXECUTED]
DEBUG:experta.watchers.AGENDA:0: 'set_pii_to_default' '<f-0>'
INFO:experta.watchers.RULES:FIRE 2 set_pii_to_default: <f-0>
INFO:experta.watchers.FACTS: ==> <f-3>: Fact(has_pii='no')
INFO:experta.watchers.ACTIVATIONS: <== 'set_pii_to_default': <f-0> [EXECUTED]
INFO:experta.watchers.ACTIVATIONS: ==> 'return_result': <f-1>, <f-3>, <f-2>
DEBUG:experta.watchers.AGENDA:0: 'return_result' '<f-1>, <f-3>, <f-2>'
INFO:experta.watchers.RULES:FIRE 3 return_result: <f-1>, <f-3>, <f-2>
INFO:experta.watchers.FACTS: ==> <f-0>: InitialFact()
INFO:experta.watchers.ACTIVATIONS: ==> 'set_target_type_to_default': <f-0>
INFO:experta.watchers.ACTIVATIONS: ==> 'set_pii_to_default': <f-0>
INFO:experta.watchers.FACTS: ==> <f-1>: Fact(name='longitude', target_type='DoubleType')
INFO:experta.watchers.ACTIVATIONS: <== 'set_target_type_to_default': <f-0> [EXECUTED]
INFO:experta.watchers.ACTIVATIONS: ==> 'set_name_as_path': <f-0>, <f-1>
DEBUG:experta.watchers.AGENDA:0: 'set_pii_to_default' '<f-0>'
DEBUG:experta.watchers.AGENDA:1: 'set_name_as_path' '<f-0>, <f-1>'
INFO:experta.watchers.RULES:FIRE 1 set_name_as_path: <f-0>, <f-1>
INFO:experta.watchers.FACTS: ==> <f-2>: Fact(path='longitude')
INFO:experta.watchers.ACTIVATIONS: <== 'set_name_as_path': <f-0>, <f-1> [EXECUTED]
DEBUG:experta.watchers.AGENDA:0: 'set_pii_to_default' '<f-0>'
INFO:experta.watchers.RULES:FIRE 2 set_pii_to_default: <f-0>
INFO:experta.watchers.FACTS: ==> <f-3>: Fact(has_pii='no')
INFO:experta.watchers.ACTIVATIONS: <== 'set_pii_to_default': <f-0> [EXECUTED]
INFO:experta.watchers.ACTIVATIONS: ==> 'return_result': <f-3>, <f-2>, <f-1>
DEBUG:experta.watchers.AGENDA:0: 'return_result' '<f-3>, <f-2>, <f-1>'
INFO:experta.watchers.RULES:FIRE 3 return_result: <f-3>, <f-2>, <f-1>
INFO:experta.watchers.FACTS: ==> <f-0>: InitialFact()
INFO:experta.watchers.ACTIVATIONS: ==> 'set_target_type_to_default': <f-0>
INFO:experta.watchers.ACTIVATIONS: ==> 'set_pii_to_default': <f-0>
INFO:experta.watchers.FACTS: ==> <f-1>: Fact(name='stars', target_type='LongType', triviality=1, desc='star rating, rounded to half-stars')
INFO:experta.watchers.ACTIVATIONS: <== 'set_target_type_to_default': <f-0> [EXECUTED]
INFO:experta.watchers.ACTIVATIONS: ==> 'set_name_as_path': <f-0>, <f-1>
DEBUG:experta.watchers.AGENDA:0: 'set_pii_to_default' '<f-0>'
DEBUG:experta.watchers.AGENDA:1: 'set_name_as_path' '<f-0>, <f-1>'
INFO:experta.watchers.RULES:FIRE 1 set_name_as_path: <f-0>, <f-1>
INFO:experta.watchers.FACTS: ==> <f-2>: Fact(path='stars')
INFO:experta.watchers.ACTIVATIONS: <== 'set_name_as_path': <f-0>, <f-1> [EXECUTED]
DEBUG:experta.watchers.AGENDA:0: 'set_pii_to_default' '<f-0>'
INFO:experta.watchers.RULES:FIRE 2 set_pii_to_default: <f-0>
INFO:experta.watchers.FACTS: ==> <f-3>: Fact(has_pii='no')
```

```
INFO:experta.watchers.ACTIVATIONS: <== 'set_pii_to_default': <f-0>
    ↪  [EXECUTED]
INFO:experta.watchers.ACTIVATIONS: ==> 'return_result': <f-3>, <f-2>,
    ↪  <f-1>
DEBUG:experta.watchers.AGENDA:0: 'return_result' '<f-3>, <f-2>, <f-1>'
INFO:experta.watchers.RULES:FIRE 3 return_result: <f-3>, <f-2>, <f-1>
INFO:experta.watchers.FACTS: ==> <f-0>: InitialFact()
INFO:experta.watchers.ACTIVATIONS: ==> 'set_target_type_to_default': <f-0>
INFO:experta.watchers.ACTIVATIONS: ==> 'set_pii_to_default': <f-0>
INFO:experta.watchers.FACTS: ==> <f-1>: Fact(name='review_count',
    ↪  target_type='LongType', triviality=1)
INFO:experta.watchers.ACTIVATIONS: <== 'set_target_type_to_default': <f-0>
    ↪  [EXECUTED]
INFO:experta.watchers.ACTIVATIONS: ==> 'set_name_as_path': <f-1>, <f-0>
DEBUG:experta.watchers.AGENDA:0: 'set_pii_to_default' '<f-0>'
DEBUG:experta.watchers.AGENDA:1: 'set_name_as_path' '<f-1>, <f-0>'
INFO:experta.watchers.RULES:FIRE 1 set_name_as_path: <f-1>, <f-0>
INFO:experta.watchers.FACTS: ==> <f-2>: Fact(path='review_count')
INFO:experta.watchers.ACTIVATIONS: <== 'set_name_as_path': <f-1>, <f-0>
    ↪  [EXECUTED]
DEBUG:experta.watchers.AGENDA:0: 'set_pii_to_default' '<f-0>'
INFO:experta.watchers.RULES:FIRE 2 set_pii_to_default: <f-0>
INFO:experta.watchers.FACTS: ==> <f-3>: Fact(has_pii='no')
INFO:experta.watchers.ACTIVATIONS: <== 'set_pii_to_default': <f-0>
    ↪  [EXECUTED]
INFO:experta.watchers.ACTIVATIONS: ==> 'return_result': <f-3>, <f-1>,
    ↪  <f-2>
DEBUG:experta.watchers.AGENDA:0: 'return_result' '<f-3>, <f-1>, <f-2>'
INFO:experta.watchers.RULES:FIRE 3 return_result: <f-3>, <f-1>, <f-2>
INFO:experta.watchers.FACTS: ==> <f-0>: InitialFact()
INFO:experta.watchers.ACTIVATIONS: ==> 'set_target_type_to_default': <f-0>
INFO:experta.watchers.ACTIVATIONS: ==> 'set_pii_to_default': <f-0>
INFO:experta.watchers.FACTS: ==> <f-1>: Fact(name='categories',
    ↪  triviality=1, target_type='json_string')
INFO:experta.watchers.ACTIVATIONS: <== 'set_target_type_to_default': <f-0>
    ↪  [EXECUTED]
INFO:experta.watchers.ACTIVATIONS: ==> 'set_name_as_path': <f-1>, <f-0>
DEBUG:experta.watchers.AGENDA:0: 'set_pii_to_default' '<f-0>'
DEBUG:experta.watchers.AGENDA:1: 'set_name_as_path' '<f-1>, <f-0>'
INFO:experta.watchers.RULES:FIRE 1 set_name_as_path: <f-1>, <f-0>
INFO:experta.watchers.FACTS: ==> <f-2>: Fact(path='categories')
INFO:experta.watchers.ACTIVATIONS: <== 'set_name_as_path': <f-1>, <f-0>
    ↪  [EXECUTED]
DEBUG:experta.watchers.AGENDA:0: 'set_pii_to_default' '<f-0>'
INFO:experta.watchers.RULES:FIRE 2 set_pii_to_default: <f-0>
INFO:experta.watchers.FACTS: ==> <f-3>: Fact(has_pii='no')
INFO:experta.watchers.ACTIVATIONS: <== 'set_pii_to_default': <f-0>
    ↪  [EXECUTED]
INFO:experta.watchers.ACTIVATIONS: ==> 'return_result': <f-3>, <f-1>,
    ↪  <f-2>
DEBUG:experta.watchers.AGENDA:0: 'return_result' '<f-3>, <f-1>, <f-2>'
INFO:experta.watchers.RULES:FIRE 3 return_result: <f-3>, <f-1>, <f-2>
INFO:experta.watchers.FACTS: ==> <f-0>: InitialFact()
INFO:experta.watchers.ACTIVATIONS: ==> 'set_target_type_to_default': <f-0>
INFO:experta.watchers.ACTIVATIONS: ==> 'set_pii_to_default': <f-0>
INFO:experta.watchers.FACTS: ==> <f-1>: Fact(name='open_on_monday',
    ↪  path='hours.Monday', triviality=10)
DEBUG:experta.watchers.AGENDA:0: 'set_target_type_to_default' '<f-0>'
DEBUG:experta.watchers.AGENDA:1: 'set_pii_to_default' '<f-0>'
INFO:experta.watchers.RULES:FIRE 1 set_pii_to_default: <f-0>
```

```
INFO:experta.watchers.FACTS: ==> <f-2>: Fact(has_pii='no')
INFO:experta.watchers.ACTIVATIONS: <== 'set_pii_to_default': <f-0>
    ↪  [EXECUTED]
DEBUG:experta.watchers.AGENDA:0: 'set_target_type_to_default' '<f-0>'
INFO:experta.watchers.RULES:FIRE 2 set_target_type_to_default: <f-0>
INFO:experta.watchers.FACTS: ==> <f-3>: Fact(target_type='StringType')
INFO:experta.watchers.ACTIVATIONS: <== 'set_target_type_to_default': <f-0>
    ↪  [EXECUTED]
INFO:experta.watchers.ACTIVATIONS: ==> 'return_result': <f-3>, <f-1>,
    ↪  <f-2>
DEBUG:experta.watchers.AGENDA:0: 'return_result' '<f-3>, <f-1>, <f-2>'
INFO:experta.watchers.RULES:FIRE 3 return_result: <f-3>, <f-1>, <f-2>
INFO:experta.watchers.FACTS: ==> <f-0>: InitialFact()
INFO:experta.watchers.ACTIVATIONS: ==> 'set_target_type_to_default': <f-0>
INFO:experta.watchers.ACTIVATIONS: ==> 'set_pii_to_default': <f-0>
INFO:experta.watchers.FACTS: ==> <f-1>: Fact(name='open_on_tuesday',
    ↪  path='hours.Tuesday', triviality=10)
DEBUG:experta.watchers.AGENDA:0: 'set_target_type_to_default' '<f-0>'
DEBUG:experta.watchers.AGENDA:1: 'set_pii_to_default' '<f-0>'
INFO:experta.watchers.RULES:FIRE 1 set_pii_to_default: <f-0>
INFO:experta.watchers.FACTS: ==> <f-2>: Fact(has_pii='no')
INFO:experta.watchers.ACTIVATIONS: <== 'set_pii_to_default': <f-0>
    ↪  [EXECUTED]
DEBUG:experta.watchers.AGENDA:0: 'set_target_type_to_default' '<f-0>'
INFO:experta.watchers.RULES:FIRE 2 set_target_type_to_default: <f-0>
INFO:experta.watchers.FACTS: ==> <f-3>: Fact(target_type='StringType')
INFO:experta.watchers.ACTIVATIONS: <== 'set_target_type_to_default': <f-0>
    ↪  [EXECUTED]
INFO:experta.watchers.ACTIVATIONS: ==> 'return_result': <f-3>, <f-1>,
    ↪  <f-2>
DEBUG:experta.watchers.AGENDA:0: 'return_result' '<f-3>, <f-1>, <f-2>'
INFO:experta.watchers.RULES:FIRE 3 return_result: <f-3>, <f-1>, <f-2>
INFO:experta.watchers.FACTS: ==> <f-0>: InitialFact()
INFO:experta.watchers.ACTIVATIONS: ==> 'set_target_type_to_default': <f-0>
INFO:experta.watchers.ACTIVATIONS: ==> 'set_pii_to_default': <f-0>
INFO:experta.watchers.FACTS: ==> <f-1>: Fact(name='open_on_wednesday',
    ↪  path='hours.Wednesday', triviality=10)
DEBUG:experta.watchers.AGENDA:0: 'set_target_type_to_default' '<f-0>'
DEBUG:experta.watchers.AGENDA:1: 'set_pii_to_default' '<f-0>'
INFO:experta.watchers.RULES:FIRE 1 set_pii_to_default: <f-0>
INFO:experta.watchers.FACTS: ==> <f-2>: Fact(has_pii='no')
INFO:experta.watchers.ACTIVATIONS: <== 'set_pii_to_default': <f-0>
    ↪  [EXECUTED]
DEBUG:experta.watchers.AGENDA:0: 'set_target_type_to_default' '<f-0>'
INFO:experta.watchers.RULES:FIRE 2 set_target_type_to_default: <f-0>
INFO:experta.watchers.FACTS: ==> <f-3>: Fact(target_type='StringType')
INFO:experta.watchers.ACTIVATIONS: <== 'set_target_type_to_default': <f-0>
    ↪  [EXECUTED]
INFO:experta.watchers.ACTIVATIONS: ==> 'return_result': <f-3>, <f-1>,
    ↪  <f-2>
DEBUG:experta.watchers.AGENDA:0: 'return_result' '<f-3>, <f-1>, <f-2>'
INFO:experta.watchers.RULES:FIRE 3 return_result: <f-3>, <f-1>, <f-2>
INFO:experta.watchers.FACTS: ==> <f-0>: InitialFact()
INFO:experta.watchers.ACTIVATIONS: ==> 'set_target_type_to_default': <f-0>
INFO:experta.watchers.ACTIVATIONS: ==> 'set_pii_to_default': <f-0>
INFO:experta.watchers.FACTS: ==> <f-1>: Fact(name='open_on_thursday',
    ↪  path='hours.Thursday', triviality=10)
DEBUG:experta.watchers.AGENDA:0: 'set_target_type_to_default' '<f-0>'
DEBUG:experta.watchers.AGENDA:1: 'set_pii_to_default' '<f-0>'
INFO:experta.watchers.RULES:FIRE 1 set_pii_to_default: <f-0>
```

```
INFO:experta.watchers.FACTS: ==> <f-2>: Fact(has_pii='no')
INFO:experta.watchers.ACTIVATIONS: <== 'set_pii_to_default': <f-0>
    ↪  [EXECUTED]
DEBUG:experta.watchers.AGENDA:0: 'set_target_type_to_default' '<f-0>'
INFO:experta.watchers.RULES:FIRE 2 set_target_type_to_default: <f-0>
INFO:experta.watchers.FACTS: ==> <f-3>: Fact(target_type='StringType')
INFO:experta.watchers.ACTIVATIONS: <== 'set_target_type_to_default': <f-0>
    ↪  [EXECUTED]
INFO:experta.watchers.ACTIVATIONS: ==> 'return_result': <f-3>, <f-1>,
    ↪  <f-2>
DEBUG:experta.watchers.AGENDA:0: 'return_result' '<f-3>, <f-1>, <f-2>'
INFO:experta.watchers.RULES:FIRE 3 return_result: <f-3>, <f-1>, <f-2>
INFO:experta.watchers.FACTS: ==> <f-0>: InitialFact()
INFO:experta.watchers.ACTIVATIONS: ==> 'set_target_type_to_default': <f-0>
INFO:experta.watchers.ACTIVATIONS: ==> 'set_pii_to_default': <f-0>
INFO:experta.watchers.FACTS: ==> <f-1>: Fact(name='open_on_friday',
    ↪  path='hours.Friday', triviality=10)
DEBUG:experta.watchers.AGENDA:0: 'set_target_type_to_default' '<f-0>'
DEBUG:experta.watchers.AGENDA:1: 'set_pii_to_default' '<f-0>'
INFO:experta.watchers.RULES:FIRE 1 set_pii_to_default: <f-0>
INFO:experta.watchers.FACTS: ==> <f-2>: Fact(has_pii='no')
INFO:experta.watchers.ACTIVATIONS: <== 'set_pii_to_default': <f-0>
    ↪  [EXECUTED]
DEBUG:experta.watchers.AGENDA:0: 'set_target_type_to_default' '<f-0>'
INFO:experta.watchers.RULES:FIRE 2 set_target_type_to_default: <f-0>
INFO:experta.watchers.FACTS: ==> <f-3>: Fact(target_type='StringType')
INFO:experta.watchers.ACTIVATIONS: <== 'set_target_type_to_default': <f-0>
    ↪  [EXECUTED]
INFO:experta.watchers.ACTIVATIONS: ==> 'return_result': <f-3>, <f-1>,
    ↪  <f-2>
DEBUG:experta.watchers.AGENDA:0: 'return_result' '<f-3>, <f-1>, <f-2>'
INFO:experta.watchers.RULES:FIRE 3 return_result: <f-3>, <f-1>, <f-2>
INFO:experta.watchers.FACTS: ==> <f-0>: InitialFact()
INFO:experta.watchers.ACTIVATIONS: ==> 'set_target_type_to_default': <f-0>
INFO:experta.watchers.ACTIVATIONS: ==> 'set_pii_to_default': <f-0>
INFO:experta.watchers.FACTS: ==> <f-1>: Fact(name='open_on_saturday',
    ↪  path='hours.Saturday', triviality=10)
DEBUG:experta.watchers.AGENDA:0: 'set_target_type_to_default' '<f-0>'
DEBUG:experta.watchers.AGENDA:1: 'set_pii_to_default' '<f-0>'
INFO:experta.watchers.RULES:FIRE 1 set_pii_to_default: <f-0>
INFO:experta.watchers.FACTS: ==> <f-2>: Fact(has_pii='no')
INFO:experta.watchers.ACTIVATIONS: <== 'set_pii_to_default': <f-0>
    ↪  [EXECUTED]
DEBUG:experta.watchers.AGENDA:0: 'set_target_type_to_default' '<f-0>'
INFO:experta.watchers.RULES:FIRE 2 set_target_type_to_default: <f-0>
INFO:experta.watchers.FACTS: ==> <f-3>: Fact(target_type='StringType')
INFO:experta.watchers.ACTIVATIONS: <== 'set_target_type_to_default': <f-0>
    ↪  [EXECUTED]
INFO:experta.watchers.ACTIVATIONS: ==> 'return_result': <f-3>, <f-1>,
    ↪  <f-2>
DEBUG:experta.watchers.AGENDA:0: 'return_result' '<f-3>, <f-1>, <f-2>'
INFO:experta.watchers.RULES:FIRE 3 return_result: <f-3>, <f-1>, <f-2>
INFO:experta.watchers.FACTS: ==> <f-0>: InitialFact()
INFO:experta.watchers.ACTIVATIONS: ==> 'set_target_type_to_default': <f-0>
INFO:experta.watchers.ACTIVATIONS: ==> 'set_pii_to_default': <f-0>
INFO:experta.watchers.FACTS: ==> <f-1>: Fact(name='attributes',
    ↪  path='attributes', target_type='json_string', triviality=10)
INFO:experta.watchers.ACTIVATIONS: <== 'set_target_type_to_default': <f-0>
    ↪  [EXECUTED]
DEBUG:experta.watchers.AGENDA:0: 'set_pii_to_default' '<f-0>'
```

```
INFO:experta.watchers.RULES:FIRE 1 set_pii_to_default: <f-0>
INFO:experta.watchers.FACTS: ==> <f-2>: Fact(has_pii='no')
INFO:experta.watchers.ACTIVATIONS: <== 'set_pii_to_default': <f-0>
↪ [EXECUTED]
INFO:experta.watchers.ACTIVATIONS: ==> 'return_result': <f-1>, <f-2>
DEBUG:experta.watchers.AGENDA:0: 'return_result' '<f-1>, <f-2>'
INFO:experta.watchers.RULES:FIRE 2 return_result: <f-1>, <f-2>
INFO:werkzeug:127.0.0.1 - - [18/Mar/2020 00:45:34] "POST
↪ /transformer/params/Mapper HTTP/1.1" 200 -
DEBUG:experta.watchers.AGENDA:0: 'return_result' '<f-1>'
INFO:experta.watchers.RULES:FIRE 1 return_result: <f-1>
INFO:werkzeug:127.0.0.1 - - [18/Mar/2020 00:45:34] "POST
↪ /transformer/params/ThresholdCleaner HTTP/1.1" 200 -
DEBUG:experta.watchers.AGENDA:0: 'hive_extractor' '<f-1>'
DEBUG:experta.watchers.AGENDA:1: 'by_passing_loader' '<f-1>'
INFO:experta.watchers.RULES:FIRE 1 by_passing_loader: <f-1>
INFO:werkzeug:127.0.0.1 - - [18/Mar/2020 00:45:34] "POST /loader/name
↪ HTTP/1.1" 200 -
DEBUG:experta.watchers.AGENDA:0: 'return_to_sender' '<f-1>'
INFO:experta.watchers.RULES:FIRE 1 return_to_sender: <f-1>
INFO:werkzeug:127.0.0.1 - - [18/Mar/2020 00:45:34] "POST
↪ /loader/params/ByPass HTTP/1.1" 200 -
INFO:werkzeug:127.0.0.1 - - [18/Mar/2020 00:45:34] "POST /pipeline/get
↪ HTTP/1.1" 200 -
```

Appendix E: Demonstration of Evolvability

Adding NoIdDropper Transformer

src/spooq2/transformer/__init__.py

```
--- original
+++ adapted
@@ -1,13 +1,15 @@
 from newest_by_group import NewestByGroup
 from mapper import Mapper
 from exploder import Exploder
 from threshold_cleaner import ThresholdCleaner
 from sieve import Sieve
+from no_id_dropper import NoIdDropper

 __all__ = [
     "NewestByGroup",
     "Mapper",
     "Exploder",
     "ThresholdCleaner",
     "Sieve",
+    "NoIdDropper",
 ]
```

tests/unit/transformer/test_no_id_dropper.py

```
import pytest
from pyspark.sql.dataframe import DataFrame

from spooq2.transformer import NoIdDropper


@pytest.fixture()
def default_transformer():
    return NoIdDropper(id_columns=["first_name", "last_name"])


@pytest.fixture()
```

```
def input_df(spark_session):
    return spark_session.read.parquet("../data/schema_v1/parquetFiles")


@pytest.fixture()
def transformed_df(default_transformer, input_df):
    return default_transformer.transform(input_df)


class TestBasicAttributes(object):

    def test_logger_should_be_accessible(self, default_transformer):
        assert hasattr(default_transformer, "logger")

    def test_name_is_set(self, default_transformer):
        assert default_transformer.name == "NoIdDropper"

    def test_str_representation_is_correct(self, default_transformer):
        assert unicode(default_transformer) == "Transformer Object of
        ↪  Class NoIdDropper"

class TestNoIdDropper(object):

    def test_records_are_dropped(transformed_df, input_df):
        """Transformed DataFrame has no records with missing first_name
        ↪  and last_name"""
        assert input_df.where("first_name is null or last_name is
        ↪  null").count() > 0
        assert transformed_df.where("first_name is null or last_name is
        ↪  null").count() == 0

    def test_schema_is_unchanged(transformed_df, input_df):
        """Converted DataFrame has the expected schema"""
        assert transformed_df.schema == input_df.schema
```

docs/source/transformer/no_id_dropper.rst

```
Record Dropper if Id is missing
===============================

Some text if you like...

.. automodule:: spooq2.transformer.no_id_dropper
```

docs/source/transformer/overview.rst

```
--- original
+++ adapted
@@ -7,14 +7,15 @@
 .. toctree::

     exploder
     sieve
     mapper
     threshold_cleaner
     newest_by_group
```

```
+    no_id_dropper

 Class Diagram of Transformer Subpackage
 -----------------------------------------------
 .. uml:: ../diagrams/from_thesis/class_diagram/transformers.puml
    :caption: Class Diagram of Transformer Subpackage
```

Adding Parquet Loader

src/spooq2/loader/__init__.py

```
--- original
+++ adapted
@@ -1,7 +1,9 @@
 from loader import Loader
 from hive_loader import HiveLoader
+from parquet import ParquetLoader

 __all__ = [
     "Loader",
     "HiveLoader",
+    "ParquetLoader",
 ]
```

tests/unit/loader/test_parquet.py

```
import pytest
from pyspark.sql.dataframe import DataFrame

from spooq2.loader import ParquetLoader


@pytest.fixture(scope="module")
def output_path(tmpdir_factory):
    return str(tmpdir_factory.mktemp("parquet_output"))


@pytest.fixture(scope="module")
def default_loader(output_path):
    return ParquetLoader(
        path=output_path,
        partition_by="attributes.gender",
        explicit_partition_values=None,
        compression_codec=None
    )


@pytest.fixture(scope="module")
def input_df(spark_session):
    return spark_session.read.parquet("../data/schema_v1/parquetFiles")
```

```

@pytest.fixture(scope="module")
def loaded_df(default_loader, input_df, spark_session, output_path):
    default_loader.load(input_df)
    return spark_session.read.parquet(output_path)


class TestBasicAttributes(object):

    def test_logger_should_be_accessible(self, default_loader):
        assert hasattr(default_loader, "logger")

    def test_name_is_set(self, default_loader):
        assert default_loader.name == "ParquetLoader"

    def test_str_representation_is_correct(self, default_loader):
        assert unicode(default_loader) == "loader Object of Class
        ↪  ParquetLoader"

class TestParquetLoader(object):

    def test_count_did_not_change(loaded_df, input_df):
        """Persisted DataFrame has the same number of records than the
        ↪  input DataFrame"""
        assert input_df.count() == output_df.count() and input_df.count()
        ↪  > 0

    def test_schema_is_unchanged(loaded_df, input_df):
        """Loaded DataFrame has the same schema as the input DataFrame"""
        assert loaded.schema == input_df.schema
```

docs/source/loader/parquet.rst

```
Parquet Loader
==============================

Some text if you like...

.. automodule:: spooq2.loader.parquet
```

docs/source/loader/overview.rst

```
--- original
+++ adapted
@@ -7,4 +7,5 @@
 .. toctree::
     hive_loader
+    parquet

 Class Diagram of Loader Subpackage
```

Appendix F: Spooq Rules Source Code

app.py

spooq_rules/app.py

```
from flask import Flask, request, jsonify, make_response, current_app
from experta import Fact, watchers, watch, unwatch
import requests
import json

from spooq_rules.health_check import (
    CrossStreet,
    TrafficLight,
    DeclareNewField,
    DeclareAdditionalField
)
from spooq_rules import extractor_rules
from spooq_rules import transformer_rules
from spooq_rules import loader_rules
from spooq_rules import metadata_fetcher
from spooq_rules import context_rules


app = Flask(__name__)

# Helper Methods
def jsonify_no_content():
    response = make_response('', 204)
    response.mimetype = current_app.config['JSONIFY_MIMETYPE']
    return response

def watch_rules(callback):
        try:
            watch("ACTIVATIONS")  # Show what rules are activated
            watch("AGENDA")  # Agenda changes
            watch("RULES")  # Show what rules are triggered
            watch("FACTS")  # Show asserted and retracted facts
            callback()
        finally:
            unwatch()
```

```
36
37
38  # Context Reasoning / MetaData Fetcher
39  @app.route("/context/get", methods=["POST"])
40  def reason_over_context_vars():
41      input_dict = request.json
42      if isinstance(input_dict, str):
43          input_dict = json.loads(input_dict)
44      reasoner = context_rules.Context()
45      reasoner.reset()
46      reasoner.declare(Fact("context", **input_dict))
47      watch_rules(reasoner.run)
48      input_dict.update(reasoner.response)
49      return input_dict
50
51  # Whole Pipeline
52  @app.route("/pipeline/get", methods=["POST"])
53  def get_pipeline():
54      output_dict = {
55          "extractor": {},
56          "transformers": [],
57          "loader": {},
58          }
59      root_url = request.host_url
60
61      context_dict = requests.post(root_url + "context/get",
        ↪  json=request.json).json()
62      metadata = metadata_fetcher.get_metadata_by_context(context_dict)
63      output_dict["context_variables"] = metadata
64
65      """Extractor"""
66      extractor_name = requests.post(root_url + "extractor/name",
        ↪  json=metadata).text
67      extractor_params = requests.post(root_url + "extractor/params/" +
        ↪  extractor_name, json=metadata).json()
68      output_dict["extractor"]["name"] = extractor_name
69      output_dict["extractor"]["params"] = extractor_params
70
71      """Transformers"""
72      transformer_names = requests.post(root_url + "transformer/names",
        ↪  json=metadata).json()
73      for transformer in transformer_names:
74          output_dict["transformers"].append(
75              {"name": transformer,
76              "params": requests.post(root_url + "transformer/params/" +
                ↪  transformer, json=metadata).json()}
77          )
78
79      """Loader"""
80      loader_name = requests.post(root_url + "loader/name",
        ↪  json=metadata).text
81      loader_params = requests.post(root_url + "loader/params/" +
        ↪  loader_name, json=metadata).json()
82      output_dict["loader"]["name"] = loader_name
83      output_dict["loader"]["params"] = loader_params
84
85      return output_dict
86
87  # Extractor Names and Parameters
88  @app.route("/extractor/name", methods=["POST"])
```

```
def get_extractor_name():
    reasoner = extractor_rules.ExtractorName()
    reasoner.reset()
    reasoner.declare(Fact(**request.json))
    watch_rules(reasoner.run)
    return reasoner.response

@app.route("/extractor/params/<extractor_name>", methods=["POST"])
def get_extractor_params(extractor_name):
    reasoner = getattr(extractor_rules, extractor_name)()
    reasoner.reset()
    reasoner.declare(Fact(**request.json))
    watch_rules(reasoner.run)
    return jsonify(reasoner.response)

# Transformer Names and Parameters
@app.route("/transformer/names", methods=["POST"])
def get_transformer_names():
    reasoner = transformer_rules.TransformerNames()
    reasoner.reset()
    reasoner.declare(Fact(**request.json))
    watch_rules(reasoner.run)
    transformer_names_unordered = reasoner.response
    transformer_names_ordered = [name for _, name in
    ↪  sorted(transformer_names_unordered, key=lambda t: t[0])]
    return jsonify(transformer_names_ordered)

@app.route("/transformer/params/<transformer_name>", methods=["POST"])
def get_transformer_params(transformer_name):
    reasoner = getattr(transformer_rules, transformer_name)()
    reasoner.reset()
    f1 = Fact(**request.json)
    reasoner.declare(Fact(**request.json))
    watch_rules(reasoner.run)
    return jsonify(reasoner.response)

# Loader Names and Parameters
@app.route("/loader/name", methods=["POST"])
def get_loader_name():
    reasoner = loader_rules.LoaderName()
    reasoner.reset()
    reasoner.declare(Fact(**request.json))
    watch_rules(reasoner.run)
    return reasoner.response

@app.route("/loader/params/<loader_name>", methods=["POST"])
def get_loader_params(loader_name):
    reasoner = getattr(loader_rules, loader_name)()
    reasoner.reset()
    f1 = Fact(**request.json)
    reasoner.declare(Fact(**request.json))
    watch_rules(reasoner.run)
    return jsonify(reasoner.response)

# Health Checks and Tests
@app.route("/health_check/robot", methods=["POST"])
def robot_test():
    """
    Receives a traffic light color and passes it to the robot engine.
    Returns:
```

```
        Engine response in json format
    """
    light = request.get_json().get("light", None)
    robot = CrossStreet()
    robot.reset()
    robot.declare(TrafficLight(color=light))
    watch_rules(robot.run)
    return jsonify({"robot_response": robot.response})

@app.route("/health_check/augmented1", methods=["POST"])
def declare_new_field():
    engine = DeclareNewField()
    engine.reset()
    engine.declare(Fact(**request.json))
    watch_rules(engine.run)
    return jsonify({"rules triggered": engine.response})

@app.route("/health_check/augmented2", methods=["POST"])
def declare_additional_field():
    engine = DeclareAdditionalField()
    engine.reset()
    engine.declare(Fact(**request.json))
    watch_rules(engine.run)
    return jsonify({"rules triggered": engine.response})

@app.route("/health_check/hello_world", methods=["GET"])
def hello_world():
    """Returns a simple json with hello world"""
    return jsonify({"hello": "world"})


if __name__ == "__main__":
    app.run(debug=True)
```

context_rules.py

spooq_rules/context_rules.py

```
from experta import (
    KnowledgeEngine,
    Rule,
    Fact,
    OR,
    AND,
    NOT,
    W,
    L,
    MATCH,
    Field
)
import schema
import datetime
```

```
15
16
17  class Context(KnowledgeEngine):
18
19      def __init__(self):
20          super().__init__()
21          self.response = {}
22
23      @Rule(NOT(Fact(date=W())))
24      def set_date_to_yesterday(self):
25          yesterday = datetime.date.today() - datetime.timedelta(1)
26          date = yesterday.strftime('%Y-%m-%d')
27          self.response["date"] = date
28          self.declare(Fact(date=date))
29
30      @Rule(NOT(Fact(pipeline_type=W())),
31            (Fact(batch_size="no")),
32            salience=5)
33      def set_pipeline_type_according_to_no_batch_size(self):
34          pipeline_type = "ad_hoc"
35          self.response["pipeline_type"] = pipeline_type
36          self.declare(Fact(pipeline_type=pipeline_type))
37
38      @Rule(NOT(Fact(pipeline_type=W())),
39            (NOT(Fact(batch_size="no"))),
40            (Fact(batch_size=W())),
41            salience=5)
42      def set_pipeline_type_according_to_set_batch_size(self):
43          pipeline_type = "batch"
44          self.response["pipeline_type"] = pipeline_type
45          self.declare(Fact(pipeline_type=pipeline_type))
46
47      @Rule(NOT(Fact(pipeline_type=W())),
48            NOT(Fact(batch_size=W())),
49            salience=4)
50      def set_default_pipeline_type(self):
51          pipeline_type = "ad_hoc"
52          batch_size = "no"
53          self.response["pipeline_type"] = pipeline_type
54          self.response["batch_size"] = batch_size
55          self.declare(Fact(pipeline_type=pipeline_type,
            ↪  batch_size=batch_size))
56
57      @Rule(NOT(Fact(batch_size=W())),
58                Fact(pipeline_type="ad_hoc"))
59      def set_batch_size_for_ad_hoc(self):
60          batch_size = "no"
61          self.response["batch_size"] = batch_size
62          self.declare(Fact(batch_size=batch_size))
63
64      @Rule(NOT(Fact(batch_size=W())),
65                Fact(pipeline_type="batch"))
66      def set_batch_size_for_batch(self):
67          batch_size = "daily"
68          self.response["batch_size"] = batch_size
69          self.declare(Fact(batch_size=batch_size))
70
71      @Rule(NOT(Fact(time_range=W())),
72                Fact(batch_size="daily"))
73      def set_time_range_for_last_day(self):
```

```
        time_range = "last_day"
        self.response["time_range"] = time_range
        self.declare(Fact(time_range=time_range))

    @Rule(NOT(Fact(time_range=W())),
          Fact(batch_size="weekly"))
    def set_time_range_for_last_week(self):
        time_range = "last_week"
        self.response["time_range"] = time_range
        self.declare(Fact(time_range=time_range))

    @Rule(NOT(Fact(time_range=W())),
          Fact(batch_size="no"))
    def set_time_range_for_all_time(self):
        time_range = "all"
        self.response["time_range"] = time_range
        self.declare(Fact(time_range=time_range))

    @Rule(NOT(Fact(level_of_detail=W())),
          Fact(pipeline_type="ad_hoc"))
    def set_level_of_detail_for_ad_hoc(self):
        level_of_detail = "all"
        self.response["level_of_detail"] = level_of_detail
        self.declare(Fact(level_of_detail=level_of_detail))

    @Rule(NOT(Fact(level_of_detail=W())),
          Fact(pipeline_type="batch"))
    def set_level_of_detail_for_batch(self):
        level_of_detail = "std"
        self.response["level_of_detail"] = level_of_detail
        self.declare(Fact(level_of_detail=level_of_detail))

    @Rule(Fact(level_of_detail=MATCH.level_of_detail), salience=10)
    def set_integer_to_level_of_detail(self, level_of_detail):
        if level_of_detail == "all":
            level_of_detail_int = 10
        elif level_of_detail == "std":
            level_of_detail_int = 5
        elif level_of_detail == "min":
            level_of_detail_int = 1
        else:
            level_of_detail_int = 5
        self.response["level_of_detail_int"] = level_of_detail_int
        self.declare(Fact(level_of_detail_int=level_of_detail_int))
```

metadata_fetcher.py

spooq_rules/metadata_fetcher.py

```
def get_metadata_by_context(context_dict):
    if context_dict["entity_type"] == "user":
        return for_user(context_dict)
    elif context_dict["entity_type"] == "business":
        return for_business(context_dict)
    elif context_dict["entity_type"] == "checkin":
        return for_checkin(context_dict)
    elif context_dict["entity_type"] == "review":
        return for_review(context_dict)
    elif context_dict["entity_type"] == "tip":
        return for_tip(context_dict)
    else:
        raise AttributeError("No Entity Type defined or
        ↪  found!\n{cntx}".format(
            cntx=str(context_dict)
        ))

def for_user(context_dict):
    context_dict.update({
        "schema": {
            "arrays_to_explode": [
                "friends",
            ],
            "grouping_keys": ["user_id", "friend"],
            "sorting_keys": ["review_count"],
            "needs_deduplication": "yes"
        },
        "value_ranges": {
            "average_stars": {"min": 1, "max": 5}
        },
        "filter_expressions": [
            "average_stars >= 2.5",
            "isnotnull(friends_element) and friends_element <> \"None\"",
        ],
        "input": {
            "locality": "internal",
            "format": "json",
            "container": "text",
            "base_path": "user"
        },
        "output": {
            "locality": "internal",
            "format": "table",
            "partition_definitions": [
                {"column_name": "p_year"},
                {"column_name": "p_month"},
                {"column_name": "p_day"},
            ],
            "repartition_size": 10,
        },
        "mapping": [
            {
                "name": "user_id",
```

```
            "target_type": "StringType",
            "triviality": 1,
            "desc": "22 character unique user id, maps to the user in
            ↪  user.json"
        },
        {
            "name": "first_name",
            "path": "name",
            "target_type": "StringType",
            "has_pii" : "yes",
            "desc": "the user's first name - anonymized"
        },
        {
            "name": "review_count",
            "target_type": "LongType",
            "triviality": 1,
            "desc": "the number of reviews they've written"
        },
        {
            "name": "yelping_since",
            "target_type": "StringType",
            "desc": "when the user joined Yelp, formatted like
            ↪  YYYY-MM-DD"
        },
        {
            "name": "average_stars",
            "target_type": "DoubleType",
            "triviality": 1,
            "desc": "average rating of all reviews"
        },
        {
            "name": "elite_years",
            "path": "elite",
            "target_type": "json_string",
            "triviality": 5,
            "desc": "the years the user was elite"
        },
        {
            "name": "friend",
            "path": "friends_element",
            "target_type": "StringType",
            "triviality": 1,
            "desc": "the user's friend as user_ids",

        },
        {
            "name": "useful",
            "target_type": "LongType",
            "triviality": 10,
            "desc": "number of useful votes sent by the user"
        },
        {
            "name": "funny",
            "target_type": "LongType",
            "triviality": 10,
            "desc": "number of funny votes sent by the user"
        },
        {
            "name": "cool",
            "target_type": "LongType",
```

```
                "triviality": 10,
                "desc": "number of cool votes sent by the user"
            },
            {
                "name": "fans",
                "target_type": "LongType",
                "desc": "number of fans the user has"
            },
            {
                "name": "compliment_hot",
                "target_type": "LongType",
                "triviality": 10,
                "desc": "number of hot compliments received by the user"
            },
            {
                "name": "compliment_more",
                "target_type": "LongType",
                "triviality": 10,
                "desc": "number of more compliments received by the user"
            },
            {
                "name": "compliment_profile",
                "target_type": "LongType",
                "triviality": 10,
                "desc": "number of profile compliments received by the
                ↪  user"
            },
            {
                "name": "compliment_cute",
                "target_type": "LongType",
                "triviality": 10,
                "desc": "number of cute compliments received by the user"
            },
            {
                "name": "compliment_list",
                "target_type": "LongType",
                "triviality": 10,
                "desc": "number of list compliments received by the user"
            },
            {
                "name": "compliment_note",
                "target_type": "LongType",
                "triviality": 10,
                "desc": "number of note compliments received by the user"
            },
            {
                "name": "compliment_plain",
                "target_type": "LongType",
                "triviality": 10,
                "desc": "number of plain compliments received by the user"
            },
            {
                "name": "compliment_cool",
                "target_type": "LongType",
                "triviality": 10,
                "desc": "number of cool compliments received by the user"
            },
            {
                "name": "compliment_funny",
                "target_type": "LongType",
```

```
                "triviality": 10,
                "desc": "number of funny compliments received by the user"
            },
            {
                "name": "compliment_writer",
                "target_type": "LongType",
                "triviality": 10,
                "desc": "number of writer compliments received by the
                ↪  user"
            },
            {
                "name": "compliment_photos",
                "target_type": "LongType",
                "triviality": 10,
                "desc": "number of photo compliments received by the user"
            },
        ]
    })
    return context_dict

def for_business(context_dict):
    context_dict.update({
        "filter_expressions": ["is_open = 1"],
        "value_ranges": {
            "stars": {"min": 1, "max": 5},
            "latitude": {"min": -90.0, "max": 90.0},
            "longitude": {"min": -180.0, "max": 180.0},
        },
        "input": {
            "locality": "internal",
            "format": "json",
            "container": "text",
            "base_path": "business"
        },
        "output": {
            "locality": "internal",
            "format": "table",
            "partition_definitions": [
                {"column_name": "p_year"},
                {"column_name": "p_month"},
                {"column_name": "p_day"},
            ],
        },
        "mapping": [
            {
                "name": "business_id",
                "triviality": 1,
                "desc": "22 character unique string"
            },
            {
                "name": "name",
                "triviality": 1,
            },
            {
                "name": "address",
                "triviality": 10,
            },
            {
                "name": "city",
            },
```

```
            {
                "name": "state",
            },
            {
                "name": "postal_code",

            },
            {
                "name": "latitude",
                "target_type": "DoubleType",
            },
            {
                "name": "longitude",
                "target_type": "DoubleType",
            },
            {
                "name": "stars",
                "target_type": "LongType",
                "triviality": 1,
                "desc": "star rating, rounded to half-stars"
            },
            {
                "name": "review_count",
                "target_type": "LongType",
                "triviality": 1,
            },
            {
                "name": "categories",
                "triviality": 1,
                "target_type": "json_string",
            },
            {
                "name": "open_on_monday",
                "path": "hours.Monday",
                "triviality": 10,
            },
            {
                "name": "open_on_tuesday",
                "path": "hours.Tuesday",
                "triviality": 10,
            },
            {
                "name": "open_on_wednesday",
                "path": "hours.Wednesday",
                "triviality": 10,
            },
            {
                "name": "open_on_thursday",
                "path": "hours.Thursday",
                "triviality": 10,
            },
            {
                "name": "open_on_friday",
                "path": "hours.Friday",
                "triviality": 10,
            },
            {
                "name": "open_on_saturday",
                "path": "hours.Saturday",
                "triviality": 10,
```

```
            },
            {   "name": "attributes",
                "path": "attributes",
                "target_type": "json_string",
                "triviality": 10
            },
        ]
    })
    return context_dict

def for_checkin(context_dict):
    return {

    }

def for_review(context_dict):
    return {

    }

def for_tip(context_dict):
    return {

    }
```

extractor_rules.py

spooq_rules/extractor_rules.py

```
from experta import (
    KnowledgeEngine,
    Rule,
    Fact,
    OR,
    AND,
    NOT,
    W,
    L,
    MATCH,
    Field
)
import schema
import datetime
from os import path


class ExtractorName(KnowledgeEngine):

    def __init__(self):
        super().__init__()
        self.response = "No extractor found!"

```

```
    @Rule(Fact(input__locality="internal"),
          Fact(input__format="json"),
          OR(Fact(input__container="sequence"),
             Fact(input__container="text")))
    def json_extractor(self):
        self.response = "JSONExtractor"

    @Rule(Fact(input__locality="internal"),
          Fact(input__container="parquet"))
    def parquet_extractor(self):
        self.response = "ParquetExtractor"

    @Rule(Fact(input__locality="internal"),
          Fact(input__container="table"))
    def hive_extractor(self):
        self.response = "HiveExtractor"

    @Rule(Fact(input__locality="external"),
          Fact(input__container='table'))
    def jdbc_type(self):
        self.declare(Fact(input__format="jdbc"))

    @Rule(Fact(input__format="jdbc"),
          Fact(input__query=W()))
    def jdbc_manual_query_extractor(self):
        self.response = 'JDBCExtractorFullLoad'

    @Rule(Fact(input__format="jdbc"),
          NOT(Fact(input__query=W())))
    def jdbc_partitioned_extractor(self):
        self.response = 'JDBCExtractorIncremental'


class JSONExtractor(KnowledgeEngine):

    def __init__(self):
        super().__init__()
        self.response = {}

    @Rule(Fact(input__path=MATCH.input__path))
    def return_result(self, input__path):
        self.response = {'input_path': input__path}

    @Rule(Fact(time_range="all"),
          Fact(input__base_path=MATCH.input__base_path))
    def input_from_all_partitions(self, input__base_path):
        input__path = path.join(input__base_path, "*", "*", "*")
        self.declare(Fact(input={"path": input__path}))

    @Rule(Fact(time_range="last_day"),
          Fact(input__base_path=MATCH.input__base_path),
          Fact(date=MATCH.date))
    def input_from_yesterday(self, input__base_path, date):
        dt = datetime.datetime.strptime(date, "%Y-%m-%d")
        partition_path = datetime.datetime.strftime(dt,
        ↪  "p_year=%Y/p_month=%m/p_day=%d")
        input__path = path.join(input__base_path, partition_path)
        self.declare(Fact(input={"path": input__path}))

    @Rule(Fact(time_range="last_week"),
```

```
            Fact(input__base_path=MATCH.input__base_path),
            Fact(date=MATCH.date))
    def input_from_last_week(self, input__base_path, date):
        dt = datetime.datetime.strptime(date, "%Y-%m-%d")
        input__path_array = []
        for delta in range(0,7):
            day = dt - datetime.timedelta(delta)
            partition_path = datetime.datetime.strftime(day,
            ↪  "p_year=%Y/p_month=%m/p_day=%d")
            input__path_array.append(path.join(input__base_path,
            ↪  partition_path))
        self.declare(Fact(input={"path": ",".join(input__path_array)}))

class ParquetExtractor(KnowledgeEngine):

    def __init__(self):
        super().__init__()
        self.response = {"response": "Not implemented!"}


class HiveExtractor(KnowledgeEngine):

    def __init__(self):
        super().__init__()
        self.response = {"response": "Not implemented!"}


class JDBCExtractorFullLoad(KnowledgeEngine):

    def __init__(self):
        super().__init__()
        self.response = {"response": "Not implemented!"}


class JDBCExtractorIncremental(KnowledgeEngine):

    def __init__(self):
        super().__init__()
        self.response = {"response": "Not implemented!"}
```

transformer_rules.py

spooq_rules/transformer_rules.py

```
from experta import (
    KnowledgeEngine,
    Rule,
    Fact,
    OR,
    AND,
    NOT,
    TEST,
```

```
    W,
    L,
    MATCH,
    Field,
    AS
)
from experta.utils import unfreeze


class TransformerNames(KnowledgeEngine):

    def __init__(self):
        super().__init__()
        self.response = []

    @Rule(Fact(schema__arrays_to_explode=MATCH.schema__arrays_to_explode))
    def arrays_to_explode_defined(self, schema__arrays_to_explode):
        for array in schema__arrays_to_explode:
            self.response.append((5, "Exploder"))

    @Rule(Fact(filter_expressions=MATCH.filter_expressions),
          Fact(level_of_detail_int=MATCH.level_of_detail_int),
          TEST(lambda level_of_detail_int: level_of_detail_int < 10))
    def filter_expressions_provided(self, filter_expressions):
        for expression in filter_expressions:
            self.response.append((10, "Sieve"))

    @Rule(Fact(mapping=MATCH.mapping))
    def mapping_provided(self, mapping):
        self.response.append((20, "Mapper"))

    @Rule(Fact(value_ranges=MATCH.value_ranges))
    def needs_cleansing(self, value_ranges):
        self.response.append((25, "ThresholdCleaner"))

    @Rule(Fact(schema__needs_deduplication="yes"))
    def needs_deduplication(self):
        self.response.append((30, "NewestByGroup"))


class Exploder(KnowledgeEngine):

    already_applied = set()

    def __init__(self):
        super().__init__()
        self.response = {"response": "No rules triggered!"}

    @Rule(Fact(schema__arrays_to_explode=MATCH.schema__arrays_to_explode))
    def return_result(self, schema__arrays_to_explode):
        for array in set(schema__arrays_to_explode) -
        ↪  self.already_applied:
            self.response = {"path_to_array": array,
                             "exploded_elem_name": array + "_element"}
            self.already_applied.add(array)
            if len(self.already_applied) ==
            ↪  len(set(schema__arrays_to_explode)):
                self.already_applied.clear()
            break

```

```
67
68  class Sieve(KnowledgeEngine):
69
70      already_applied = set()
71
72      def __init__(self):
73          super().__init__()
74          self.response = {"response": "No rules triggered!"}
75
76      @Rule(Fact(filter_expressions=MATCH.filter_expressions))
77      def return_result(self, filter_expressions):
78          for filter_expression in set(filter_expressions) -
            ↪  self.already_applied:
79              self.response = {"filter_expression": filter_expression}
80              self.already_applied.add(filter_expression)
81              if len(self.already_applied) == len(set(filter_expressions)):
82                  self.already_applied.clear()
83              break
84
85
86  class Mapper(KnowledgeEngine):
87
88      def __init__(self):
89          super().__init__()
90          self.response = {"response": "No rules triggered!"}
91
92      @Rule(Fact(mapping=MATCH.mapping),
93            Fact(level_of_detail_int=MATCH.level_of_detail_int))
94      def reason_over_each_column_and_return_results(self, mapping,
        ↪  level_of_detail_int):
95          self.response = {"mapping": []}
96          column_engine = self.Column()
97          column_facts = [Fact(**d)
98                          for d
99                          in mapping
100                         if d.get("triviality", 10) <= level_of_detail_int]
101         for fact in column_facts:
102             column_engine.reset()
103             column_engine.declare(fact)
104             column_engine.run()
105             self.response["mapping"].append(column_engine.response)
106
107
108     class Column(KnowledgeEngine):
109
110         def __init__(self):
111             super().__init__()
112             self.response = "No rules triggered!"
113
114         @Rule(NOT(Fact(path=W())),
115               Fact(name=MATCH.name))
116         def set_name_as_path(self, name):
117             self.declare(Fact(path=name))
118
119         @Rule(NOT(Fact(target_type=W())))
120         def set_target_type_to_default(self):
121             self.declare(Fact(target_type="StringType"))
122
123         @Rule(NOT(Fact(has_pii=W())))
124         def set_pii_to_default(self):
```

```
            self.declare(Fact(has_pii="no"))

        @Rule(Fact(has_pii="yes"),
              Fact(target_type="StringType"),
              salience=10)
        def set_target_type_to_string_boolean(self):
            self.declare(Fact(target_type="StringBoolean"))

        @Rule(Fact(has_pii="yes"),
              OR(Fact(target_type="IntegerType"),
                 Fact(target_type="LongType")),
              salience=10)
        def set_target_type_to_int_boolean(self):
            self.declare(Fact(target_type="IntBoolean"))

        @Rule(Fact(has_pii="yes"),
              Fact(target_type="TimestampType"),
              salience=10)
        def set_target_type_to_timestamp_month(self):
            self.declare(Fact(target_type="TimestampMonth"))

        @Rule(Fact(name=MATCH.name),
              Fact(path=MATCH.path),
              Fact(target_type=MATCH.target_type),
              Fact(has_pii=MATCH.has_pii),
              salience=1)
        def return_result(self, name, path, target_type, has_pii):
            self.response = (name, path, target_type)
            self.halt()


class ThresholdCleaner(KnowledgeEngine):

    def __init__(self):
        super().__init__()
        self.response = {"response": "No rules triggered!"}

    @Rule(Fact(value_ranges=MATCH.value_ranges))
    def return_result(self, value_ranges):
        self.response = {"thresholds": unfreeze(value_ranges)}


class NewestByGroup(KnowledgeEngine):

    def __init__(self):
        super().__init__()
        self.response = {"response": "No rules triggered!"}

    @Rule(NOT(Fact(schema__grouping_keys=W())))
    def set_grouping_keys_to_default(self):
        self.declare(Fact(schema={"grouping_keys": ["id"]}))

    @Rule(NOT(Fact(schema__sorting_keys=W())))
    def set_sorting_keys_to_default(self):
        self.declare(Fact(schema={"sorting_keys": ["udpated_at",
        ↪  "deleted_at"]}))

    @Rule(Fact(schema__grouping_keys=MATCH.schema__grouping_keys),
          Fact(schema__sorting_keys=MATCH.schema__sorting_keys))
    def return_result(self, schema__grouping_keys, schema__sorting_keys):
```

```
184         self.response = {"group_by": schema__grouping_keys,
185                          "order_by": schema__sorting_keys}
```

loader_rules.py

spooq_rules/loader_rules.py

```
1   from experta import (
2       KnowledgeEngine,
3       Rule,
4       Fact,
5       OR,
6       AND,
7       NOT,
8       TEST,
9       W,
10      L,
11      MATCH,
12      Field,
13      AS,
14      utils
15  )
16  import datetime
17
18
19  class LoaderName(KnowledgeEngine):
20
21      def __init__(self):
22          super().__init__()
23          self.response = "No loader found!"
24
25      @Rule(Fact(pipeline_type="ad_hoc"),
26            salience=100)
27      def by_passing_loader(self):
28          self.response = "ByPass"
29          self.halt()
30
31      @Rule(Fact(output__locality="internal"),
32            Fact(output__format="json"),
33            OR(Fact(output__container="sequence"),
34               Fact(output__container="text")))
35      def json_extractor(self):
36          self.response = "JSONLoader"
37
38      @Rule(Fact(output__locality="internal"),
39            Fact(output__format="parquet"))
40      def parquet_extractor(self):
41          self.response = "ParquetLoader"
42
43      @Rule(Fact(output__locality="internal"),
44            Fact(output__format="table"))
45      def hive_extractor(self):
```

```
        self.response = "HiveLoader"

    @Rule(Fact(output__locality="external"),
          Fact(output__format='table'))
    def jdbc_type(self):
        self.declare(Fact(output__format="jdbc"))

    @Rule(Fact(output__format="jdbc"),
          Fact(output__query=W()))
    def jdbc_manual_query_extractor(self):
        self.response = 'JDBCLoaderFullLoad'

    @Rule(Fact(output__format="jdbc"),
          NOT(Fact(output__query=W())))
    def jdbc_partitioned_extractor(self):
        self.response = 'JDBCLoaderIncremental'


class ByPass(KnowledgeEngine):

    def __init__(self):
        super().__init__()
        self.response = {}

    @Rule(Fact())
    def return_to_sender(self):
        pass

class HiveLoader(KnowledgeEngine):

    def __init__(self):
        super().__init__()
        self.response = {}

    @Rule(Fact(output__partition_definitions=
    ↪  MATCH.output__partition_definitions),
          Fact(date=MATCH.date),
          salience=10)
    def fix_values_in_partition_definitions(self,
    ↪  output__partition_definitions, date):
        fixed_definitions = []
        for partition_definition in output__partition_definitions:
            partition_engine = self.PartitionDefinition()
            partition_engine.reset()
            partition_engine.declare(Fact(**partition_definition,
            ↪  date=date))
            partition_engine.run()
            fixed_definitions.append(partition_engine.response)
        self.response["partition_definitions"] = fixed_definitions

    @Rule(NOT(Fact(output__db_name=W())),
          Fact(entity_type=MATCH.entity_type))
    def set_entity_name_as_db_name_name(self, entity_type):
        self.declare(Fact(output={"db_name": entity_type}))

    @Rule(NOT(Fact(output__table_prefix=W())),
          Fact(output__db_name=MATCH.output__db_name))
    def set_table_prefix_from_db_name_name(self, output__db_name):
        self.declare(Fact(output={"table_prefix":output__db_name}))

```

```
    @Rule(NOT(Fact(output__table_name=W())),
          Fact(batch_size=MATCH.batch_size),
          Fact(output__table_prefix=MATCH.output__table_prefix))
    def set_table_name_with_prefix(self, batch_size,
    ↪  output__table_prefix):
        table_name =
        ↪  "{pre}s_{bz}_partitions".format(pre=output__table_prefix,
        ↪  bz=batch_size)
        self.declare(Fact(output={"table_name": table_name}))

    @Rule(NOT(Fact(output__clear_partition=W())))
    def set_default_for_clear_partition(self):
        self.declare(Fact(output={"clear_partition": True}))

    @Rule(NOT(Fact(output__repartition_size=W())))
    def set_default_for_repartition_size(self):
        self.declare(Fact(output={"repartition_size": 40}))

    @Rule(NOT(Fact(output__auto_create_table=W())))
    def set_default_for_auto_create_table(self):
        self.declare(Fact(output={"auto_create_table": True}))

    @Rule(NOT(Fact(output__overwrite_partition_value=W())))
    def set_default_for_overwrite_partition_value(self):
        self.declare(Fact(output={"overwrite_partition_value": True}))

    @Rule(Fact(output__db_name=MATCH.output__db_name),
          Fact(output__table_name=MATCH.output__table_name),
          Fact(output__partition_definitions=
          ↪  MATCH.output__partition_definitions),
          Fact(output__clear_partition=MATCH.output__clear_partition),
          Fact(output__repartition_size=MATCH.output__repartition_size),
          Fact(output__auto_create_table=MATCH.output__auto_create_table),
          Fact(output__overwrite_partition_value=
          ↪  MATCH.output__overwrite_partition_value))
    def return_result(
        self,
        output__db_name,
        output__table_name,
        output__partition_definitions,
        output__clear_partition,
        output__repartition_size,
        output__auto_create_table,
        output__overwrite_partition_value):
        self.response.update({
            "db_name": output__db_name,
            "table_name": output__table_name,
            "clear_partition": output__clear_partition,
            "repartition_size": output__repartition_size,
            "auto_create_table": output__auto_create_table,
            "overwrite_partition_value": output__overwrite_partition_value
        })


    class PartitionDefinition(KnowledgeEngine):

        def __init__(self):
            super().__init__()
            self.response = "No rules triggered!"

```

```
        @Rule(NOT(Fact(column_type=W())))
        def set_default_for_data_type(self):
            self.declare(Fact(column_type="IntegerType"))

        @Rule(NOT(Fact(default_value=W())),
              Fact(column_name=MATCH.column_name),
              Fact(date=MATCH.date),
              TEST(lambda column_name: "year" in column_name.lower()),
              salience=10)
        def set_default_value_to_year(self, column_name, date):
            # print("set_default_value_to_year"); import IPython;
            ↪  IPython.embed()
            dt = datetime.datetime.strptime(date, "%Y-%m-%d")
            val = dt.year
            self.declare(Fact(default_value=val))

        @Rule(NOT(Fact(default_value=W())),
              Fact(column_name=MATCH.column_name),
              Fact(date=MATCH.date),
              TEST(lambda column_name: "month" in column_name.lower()),
              salience=10)
        def set_default_value_to_month(self, column_name, date):
            # print("set_default_value_to_month"); import IPython;
            ↪  IPython.embed()
            dt = datetime.datetime.strptime(date, "%Y-%m-%d")
            val = dt.month
            self.declare(Fact(default_value=val))

        @Rule(NOT(Fact(default_value=W())),
              Fact(column_name=MATCH.column_name),
              Fact(date=MATCH.date),
              TEST(lambda column_name: "day" in column_name.lower()),
              salience=10)
        def set_default_value_to_day(self, column_name, date):
            # print("set_default_value_to_day"); import IPython;
            ↪  IPython.embed()
            dt = datetime.datetime.strptime(date, "%Y-%m-%d")
            val = dt.day
            self.declare(Fact(default_value=val))

        @Rule(NOT(Fact(default_value=W())),
              Fact(column_name=MATCH.column_name),
              salience=1)
        def set_default_value_to_none(self, column_name):
            self.declare(Fact(default_value=None))

        @Rule(Fact(column_name=MATCH.column_name),
              Fact(column_type=MATCH.column_type),
              Fact(default_value=MATCH.default_value))
        def return_result(self, column_name, column_type, default_value):
            self.response = {
                "column_name": column_name,
                "column_type": column_type,
                "default_value": default_value,
            }
```

health_check.py

spooq_rules/health_check.py

```
import experta as pk


class TrafficLight(pk.Fact):
    """
    Traffic light info
    """
    pass


class CrossStreet(pk.KnowledgeEngine):
    """
    Decide if our robot is safe to cross the road.
    """

    def __init__(self):
        super().__init__()
        self.response = 'No rules triggered!'

    @pk.Rule(TrafficLight(color='green'))
    def green_light(self):
        self.response = 'Cross the road'

    @pk.Rule(TrafficLight(color='red'))
    def red_light(self):
        self.response = 'Don\'t cross the road'

    @pk.Rule('light' << TrafficLight(color=pk.L('yellow') |
    ↪  pk.L('blinking_yellow')))
    def caution(self, light):
        self.response = 'You can cross, but be careful!'


class DeclareNewField(pk.KnowledgeEngine):
    def __init__(self):
        super().__init__()
        self.response = 'No rules triggered!'

    @pk.Rule(pk.Fact(field1='val1'))
    def rule1(self):
        self.response = 'Rule 1 triggered'
        self.declare(pk.Fact(field2='val2'))

    @pk.Rule(pk.Fact(field2='val2'))
    def rule2(self):
        self.response = 'Rule 2 triggered'


class DeclareAdditionalField(pk.KnowledgeEngine):
    def __init__(self):
        super().__init__()
        self.response = 'No rules triggered!'

```

```
53      @pk.Rule(pk.Fact(field1='val1'))
54      def rule1(self):
55          self.response = 'Rule 1 triggered'
56          self.declare(pk.Fact(field2='val2'))
57
58      @pk.Rule(pk.Fact(field1='val1'),
59               pk.Fact(field2='val2'))
60      def rule2(self):
61          self.response = 'Rule 2 triggered'
```

Appendix G: Spooq Test Output

Spooq Unit Test Output

```
============================= test session starts ==============================
platform linux2 -- Python 2.7.17, pytest-3.10.1, py-1.8.1, pluggy-0.13.1
Spark will be initialized with options:
  spark.app.name: spooq-pyspark-tests
  spark.default.parallelism: 1
  spark.driver.extraClassPath: ../bin/custom_jars/sqlite-jdbc.jar
  spark.dynamicAllocation.enabled: false
  spark.executor.cores: 1
  spark.executor.extraClassPath: ../bin/custom_jars/sqlite-jdbc.jar
  spark.executor.instances: 7
  spark.io.compression.codec: lz4
  spark.rdd.compress: false
  spark.shuffle.compress: false
  spark.sql.shuffle.partitions: 1
rootdir: /home/david/projects/spooq2/tests, inifile: pytest.ini
plugins: html-1.19.0, doubles-1.5.0, metadata-1.8.0, cov-2.5.1, env-0.6.2, pspec-0.0.3, spark-0.5.2, assume-1.2.1, mock-2.0.0
collected 361 items

unit/extractor/test_jdbc.py
Basic Attributes
  ✓ logger should be accessible
  ✓ name is set
  ✓ str representation is correct

Deriving boundaries from previous loads logs (spooq2_values_pd_df)
  ✓ Getting the upper boundary partition to load
  ✓ Getting the upper boundary partition to load
  ✓ Getting the upper boundary partition to load
  ✓ Getting the lower boundary partition to load
  ✓ Getting the lower boundary partition to load
  ✓ Getting the lower boundary partition to load
  ✓ Getting the lower boundary partition to load
  ✓ get lower and upper bounds from current partition[20180515-boundaries0]
  ✓ get lower and upper bounds from current partition[20180516-boundaries1]
  ✓ get lower and upper bounds from current partition[20180517-boundaries2]
```

```
  ✓ Getting boundaries from previously loaded partitions
  ✓ get boundaries for import[20180510-boundaries0]
  ✓ get boundaries for import[20180515-boundaries1]
  ✓ get boundaries for import[20180516-boundaries2]
  ✓ get boundaries for import[20180517-boundaries3]
  ✓ get boundaries for import[20180518-boundaries4]
  ✓ get boundaries for import[20180520-boundaries5]

Constructing Query for Source Extraction with Boundaries in Where Clause
  ✓ construct query for partition[boundaries0-select * from MOCK DATA]
  ✓ construct query for partition[boundaries1-select * from MOCK DATA
↪  where updated at <= 1024]
  ✓ construct query for partition[boundaries2-select * from MOCK DATA
↪  where updated at <= 1024]
  ✓ construct query for partition[boundaries3-select * from MOCK DATA
↪  where updated at <= "g1024"]
  ✓ construct query for partition[boundaries4-select * from MOCK DATA
↪  where updated at <= "2018-05-16 03:29:59"]
  ✓ construct query for partition[boundaries5-select * from MOCK DATA
↪  where updated at > 1024]
  ✓ construct query for partition[boundaries6-select * from MOCK DATA
↪  where updated at > 1024]
  ✓ construct query for partition[boundaries7-select * from MOCK DATA
↪  where updated at > "g1024"]
  ✓ construct query for partition[boundaries8-select * from MOCK DATA
↪  where updated at > "2018-05-16 03:29:59"]
  ✓ construct query for partition[boundaries9-select * from MOCK DATA
↪  where updated at > "2018-01-01 03:30:00" and updated at <= "2018-05-16
↪  03:29:59"]

JDBC Options
  ✓ missing jdbc option raises error[url]
  ✓ missing jdbc option raises error[driver]
  ✓ missing jdbc option raises error[user]
  ✓ missing jdbc option raises error[password]
  ✓ wrong jdbc option raises error[url]
  ✓ wrong jdbc option raises error[driver]
  ✓ wrong jdbc option raises error[user]
  ✓ wrong jdbc option raises error[password]

unit/extractor/test_json_files.py
Basic Attributes
  ✓ logger should be accessible
  ✓ name is set
  ✓ str representation is correct

Path manipulating Methods
  ✓ infer input path from partition[input params0-base/17/06/01/*]
  ✓ infer input path from partition[input params1-/base/17/06/01/*]
  ✓ infer input path from partition[input params2-/base/path/17/06/01/*]
  ✓ infer input path from partition[input params3-/base/path/17/06/01/*]
  ✓ Chooses whether to use Full Input Path or derive it from Base Path
↪  and Partition
  ✓ Chooses whether to use Full Input Path or derive it from Base Path
↪  and Partition
  ✓ Chooses whether to use Full Input Path or derive it from Base Path
↪  and Partition

Extraction of JSON Files
  ✓ JSON File is converted to a DataFrame
```

```
    ✓ JSON File is converted to a DataFrame
    ✓ JSON File is converted to the correct schema
    ✓ JSON File is converted to the correct schema
    ✓ Converted DataFrame contains the same Number of Rows as in the Source
↪   Data
    ✓ Converted DataFrame contains the same Number of Rows as in the Source
↪   Data

unit/loader/test_hive_loader.py
Basic Attributes
    ✓ logger should be accessible
    ✓ name is set
    ✓ str representation is correct

Warnings
    ✓ more columns than expected
    ✓ less columns than expected
    ✓ different columns order than expected

Clearing the Hive Table Partition before inserting
    ✓ Partition is dropped
    ✓ Partition is dropped
    ✓ Partition is dropped
    ✓ Partition is dropped
    ✓ Partition is dropped
    ✓ Clear Partition is called exactly once (Default)
    ✓ Clear Partition is not called (Default Values was Overridden)

Partition Definitions
    ✓ input is not a list[Some string]
    ✓ input is not a list[123]
    ✓ input is not a list[75.0]
    ✓ input is not a list[abcd]
    ✓ input is not a list[partition definitions4]
    ✓ input is not a list[partition definitions5]
    ✓ list input contains non dict items[Some string]
    ✓ list input contains non dict items[123]
    ✓ list input contains non dict items[75.0]
    ✓ list input contains non dict items[abcd]
    ✓ list input contains non dict items[partition definitions4]
    ✓ column name is missing
    ✓ column type not a valid spark sql type[13]
    ✓ column type not a valid spark sql type[no spark type]
    ✓ column type not a valid spark sql type[arrray]
    ✓ column type not a valid spark sql type[INT]
    ✓ column type not a valid spark sql type[data type4]
    ✓ default value is empty[None]
    ✓ default value is empty[]
    ✓ default value is empty[default value2]
    ✓ default value is empty[default value3]
    ✓ default value is missing

Load Partition
    ✓ add new static partition[0]
    ✓ add new static partition[2]
    ✓ add new static partition[3]
    ✓ add new static partition[6]
    ✓ add new static partition[9]
    ✓ overwrite static partition[0]
    ✓ overwrite static partition[2]
```

```
    ✓ overwrite static partition[3]
    ✓ overwrite static partition[6]
    ✓ overwrite static partition[9]
    ✓ append to static partition[0]
    ✓ append to static partition[2]
    ✓ append to static partition[3]
    ✓ append to static partition[6]
    ✓ append to static partition[9]
    ✓ create partitioned table[0]
    ✓ create partitioned table[2]
    ✓ create partitioned table[3]
    ✓ create partitioned table[6]
    ✓ create partitioned table[9]
    ✓ add new static partition with overwritten partition value[0]
    ✓ add new static partition with overwritten partition value[2]
    ✓ add new static partition with overwritten partition value[3]
    ✓ add new static partition with overwritten partition value[6]
    ✓ add new static partition with overwritten partition value[9]

  Clearing the Hive Table Partition before inserting
    ✓ Partition is dropped
    ✓ Partition is dropped
    ✓ Partition is dropped
    ✓ Partition is dropped
    ✓ Partition is dropped
    ✓ Clear Partition is called exactly once (Default)
    ✓ Clear Partition is not called (Default Values was Overridden)

  Partition Definitions
    ✓ input is not a list[Some string]
    ✓ input is not a list[123]
    ✓ input is not a list[75.0]
    ✓ input is not a list[abcd]
    ✓ input is not a list[partition definitions4]
    ✓ input is not a list[partition definitions5]
    ✓ list input contains non dict items[Some string]
    ✓ list input contains non dict items[123]
    ✓ list input contains non dict items[75.0]
    ✓ list input contains non dict items[abcd]
    ✓ list input contains non dict items[partition definitions4]
    ✓ column name is missing
    ✓ column type not a valid spark sql type[13]
    ✓ column type not a valid spark sql type[no spark type]
    ✓ column type not a valid spark sql type[arrray]
    ✓ column type not a valid spark sql type[INT]
    ✓ column type not a valid spark sql type[data type4]
    ✓ default value is empty[None]
    ✓ default value is empty[]
    ✓ default value is empty[default value2]
    ✓ default value is empty[default value3]
    ✓ default value is missing

  Load Partition
    ✓ add new static partition[partition0]
    ✓ add new static partition[partition1]
    ✓ add new static partition[partition2]
    ✓ add new static partition[partition3]
    ✓ add new static partition[partition4]
    ✓ overwrite static partition[partition0]
    ✓ overwrite static partition[partition1]
```

```
    ✓ overwrite static partition[partition2]
    ✓ overwrite static partition[partition3]
    ✓ overwrite static partition[partition4]
    ✓ append to static partition[partition0]
    ✓ append to static partition[partition1]
    ✓ append to static partition[partition2]
    ✓ append to static partition[partition3]
    ✓ append to static partition[partition4]
    ✓ create partitioned table[partition0]
    ✓ create partitioned table[partition1]
    ✓ create partitioned table[partition2]
    ✓ create partitioned table[partition3]
    ✓ create partitioned table[partition4]
    ✓ add new static partition with overwritten partition value[partition0]
    ✓ add new static partition with overwritten partition value[partition1]
    ✓ add new static partition with overwritten partition value[partition2]
    ✓ add new static partition with overwritten partition value[partition3]
    ✓ add new static partition with overwritten partition value[partition4]

  unit/pipeline/test_pipeline.py
  Pipeline with an Extractor, a Transformers and a Loader
    ✓ Extracting from JSON SequenceFile, Mapping and Loading to Hive Table

  unit/pipeline/test_pipeline_factory.py
  ETL Batch Pipeline
    ✓ get pipeline[ETL Batch Pipeline]
    ✓ get pipeline[ELT Ad Hoc Pipeline]

  unit/transformer/test_exploder.py
  Mapper for Exploding Arrays
    ✓ logger should be accessible
    ✓ name is set
    ✓ str representation is correct

  Exploding
    ✓ count
    ✓ exploded array is added
    ✓ array is converted to struct

  unit/transformer/test_mapper.py
  Basic attributes and parameters
    ✓ logger
    ✓ name
    ✓ str representation

  Shape Of Mapped Data Frame
    ✓ Amount of Rows is the same after the transformation
    ✓ Amount of Columns of the mapped DF is according to the Mapping
    ✓ Mapped DF has renamed the Columns according to the Mapping
    ✓ base column is missing in input
    ✓ struct column is empty in input

  Data Types Of Mapped Data Frame
    ✓ data type of mapped column[id-integer]
    ✓ data type of mapped column[guid-string]
    ✓ data type of mapped column[created at-long]
    ✓ data type of mapped column[created at ms-long]
    ✓ data type of mapped column[birthday-timestamp]
    ✓ data type of mapped column[location struct-struct]
    ✓ data type of mapped column[latitude-double]
```

```
    ✓ data type of mapped column[longitude-double]
    ✓ data type of mapped column[birthday str-string]
    ✓ data type of mapped column[email-string]
    ✓ data type of mapped column[myspace-string]
    ✓ data type of mapped column[first name-string]
    ✓ data type of mapped column[last name-string]
    ✓ data type of mapped column[gender-string]
    ✓ data type of mapped column[ip address-string]
    ✓ data type of mapped column[university-string]
    ✓ data type of mapped column[friends-array]
    ✓ data type of mapped column[friends json-string]

  unit/transformer/test_mapper_custom_data_types.py
  Dynamically Call Methods By Data Type Name
    ✓ get select expression for custom type[ generate select expression for
  ↪  as is-as is]
    ✓ get select expression for custom type[ generate select expression
  ↪  without casting-as is]
    ✓ get select expression for custom type[ generate select expression
  ↪  without casting-keep]
    ✓ get select expression for custom type[ generate select expression
  ↪  without casting-no change]
    ✓ get select expression for custom type[ generate select expression for
  ↪  json string-json string]
    ✓ get select expression for custom type[ generate select expression for
  ↪  timestamp ms to ms-timestamp ms to ms]
    ✓ get select expression for custom type[ generate select expression for
  ↪  timestamp ms to s-timestamp ms to s]
    ✓ get select expression for custom type[ generate select expression for
  ↪  timestamp s to ms-timestamp s to ms0]
    ✓ get select expression for custom type[ generate select expression for
  ↪  timestamp s to ms-timestamp s to ms1]
    ✓ get select expression for custom type[ generate select expression for
  ↪  StringNull-StringNull]
    ✓ get select expression for custom type[ generate select expression for
  ↪  IntNull-IntNull]
    ✓ get select expression for custom type[ generate select expression for
  ↪  IntBoolean-IntBoolean]
    ✓ get select expression for custom type[ generate select expression for
  ↪  StringBoolean-StringBoolean]
    ✓ get select expression for custom type[ generate select expression for
  ↪  TimestampMonth-TimestampMonth]
    ✓ exception is raised if data type not found

  mapper custom data types
    ✓ generate select expression without casting[only some text-only some
  ↪  text]
    ✓ generate select expression without casting[None-None]
    ✓ generate select expression without casting[input value2-value2]
    ✓ generate select expression without casting[input value3-value3]
    ✓ generate select expression without casting[input value4-value4]
    ✓ generate select expression without casting[input value5-value5]
    ✓ generate select expression without casting[input value6-value6]
    ✓ generate select expression without casting[input value7-value7]
    ✓ generate select expression for json string[only some text-only some
  ↪  text]
    ✓ generate select expression for json string[None-None]
    ✓ generate select expression for json string[input value2-{"key":
  ↪  "value"}]
```

```
    ✓ generate select expression for json string[input value3-{"key":
↪  {"other key": "value"}}]
    ✓ generate select expression for json string[input value4-{"age": 18,
↪  "weight": 75}]
    ✓ generate select expression for json string[input value5-{"list of
↪  friend ids": [12, 75, 44, 76]}]
    ✓ generate select expression for json string[input value6-[{"weight":
↪  "75"}, {"weight": "76"}, {"weight": "73"}]]
    ✓ generate select expression for json string[input value7-{"list of
↪  friend ids": [{"id": 12}, {"id": 75}, {"id": 44}, {"id": 76}]}]

Anonymizing Methods
    ✓ generate select expression for StringBoolean[my first
↪  mail@myspace.com-1]
    ✓ generate select expression for StringBoolean[-None]
    ✓ generate select expression for StringBoolean[None-None]
    ✓ generate select expression for StringBoolean[ -1]
    ✓ generate select expression for StringBoolean[100-1]
    ✓ generate select expression for StringBoolean[0-1]
    ✓ generate select expression for StringNull[my first
↪  mail@myspace.com-None]
    ✓ generate select expression for StringNull[-None]
    ✓ generate select expression for StringNull[None-None]
    ✓ generate select expression for StringNull[ -None]
    ✓ generate select expression for StringNull[100-None]
    ✓ generate select expression for StringNull[0-None]
    ✓ generate select expression for IntBoolean[12345-1]
    ✓ generate select expression for IntBoolean[-1]
    ✓ generate select expression for IntBoolean[some text-1]
    ✓ generate select expression for IntBoolean[None-None]
    ✓ generate select expression for IntBoolean[0-1]
    ✓ generate select expression for IntBoolean[1-1]
    ✓ generate select expression for IntBoolean[-1-1]
    ✓ generate select expression for IntBoolean[5445.23-1]
    ✓ generate select expression for IntBoolean[inf-1]
    ✓ generate select expression for IntBoolean[-inf-1]
    ✓ generate select expression for IntNull[12345-None]
    ✓ generate select expression for IntNull[-None]
    ✓ generate select expression for IntNull[some text-None]
    ✓ generate select expression for IntNull[None-None]
    ✓ generate select expression for IntNull[0-None]
    ✓ generate select expression for IntNull[1-None]
    ✓ generate select expression for IntNull[-1-None]
    ✓ generate select expression for IntNull[5445.23-None]
    ✓ generate select expression for IntNull[inf-None]
    ✓ generate select expression for IntNull[-inf-None]
    ✓ generate select expression for TimestampMonth[None-None]
    ✓ generate select expression for TimestampMonth[1955-09-41-None]
    ✓ generate select expression for TimestampMonth[1969-04-03-1969-04-01]
    ✓ generate select expression for TimestampMonth[1985-03-07-1985-03-01]
    ✓ generate select expression for TimestampMonth[1998-06-10-1998-06-01]
    ✓ generate select expression for TimestampMonth[1967-05-16-1967-05-01]
    ✓ generate select expression for TimestampMonth[1953-01-01-1953-01-01]
    ✓ generate select expression for TimestampMonth[1954-11-06-1954-11-01]
    ✓ generate select expression for TimestampMonth[1978-09-05-1978-09-01]
    ✓ generate select expression for TimestampMonth[1999-05-23-1999-05-01]

Timestamp Methods
    ✓ generate select expression for timestamp ms to ms[0-0]
    ✓ generate select expression for timestamp ms to ms[-1-None]
```

```
  ✓ generate select expression for timestamp ms to ms[None-None]
  ✓ generate select expression for timestamp ms to
↪  ms[4102358400000-4102358400000]
  ✓ generate select expression for timestamp ms to ms[4102358400001-None]
  ✓ generate select expression for timestamp ms to ms[5049688276000-None]
  ✓ generate select expression for timestamp ms to
↪  ms[3469296996000-3469296996000]
  ✓ generate select expression for timestamp ms to ms[7405162940000-None]
  ✓ generate select expression for timestamp ms to
↪  ms[2769601503000-2769601503000]
  ✓ generate select expression for timestamp ms to
↪  ms[-1429593275000-None]
  ✓ generate select expression for timestamp ms to
↪  ms[3412549669000-3412549669000]
  ✓ generate select expression for timestamp ms to s[0-0]
  ✓ generate select expression for timestamp ms to s[-1-None]
  ✓ generate select expression for timestamp ms to s[None-None]
  ✓ generate select expression for timestamp ms to
↪  s[4102358400000-4102358400]
  ✓ generate select expression for timestamp ms to s[4102358400001-None]
  ✓ generate select expression for timestamp ms to s[5049688276000-None]
  ✓ generate select expression for timestamp ms to
↪  s[3469296996000-3469296996]
  ✓ generate select expression for timestamp ms to s[7405162940000-None]
  ✓ generate select expression for timestamp ms to
↪  s[2769601503000-2769601503]
  ✓ generate select expression for timestamp ms to s[-1429593275000-None]
  ✓ generate select expression for timestamp ms to
↪  s[3412549669000-3412549669]
  ✓ generate select expression for timestamp s to ms[0-0]
  ✓ generate select expression for timestamp s to ms[-1-None]
  ✓ generate select expression for timestamp s to ms[None-None]
  ✓ generate select expression for timestamp s to
↪  ms[4102358400-4102358400000]
  ✓ generate select expression for timestamp s to ms[4102358401-None]
  ✓ generate select expression for timestamp s to ms[5049688276-None]
  ✓ generate select expression for timestamp s to
↪  ms[3469296996-3469296996000]
  ✓ generate select expression for timestamp s to ms[7405162940-None]
  ✓ generate select expression for timestamp s to
↪  ms[2769601503-2769601503000]
  ✓ generate select expression for timestamp s to ms[-1429593275-None]
  ✓ generate select expression for timestamp s to
↪  ms[3412549669-3412549669000]
  ✓ generate select expression for timestamp s to s[0-0]
  ✓ generate select expression for timestamp s to s[-1-None]
  ✓ generate select expression for timestamp s to s[None-None]
  ✓ generate select expression for timestamp s to
↪  s[4102358400-4102358400]
  ✓ generate select expression for timestamp s to s[4102358401-None]
  ✓ generate select expression for timestamp s to s[5049688276-None]
  ✓ generate select expression for timestamp s to
↪  s[3469296996-3469296996]
  ✓ generate select expression for timestamp s to s[7405162940-None]
  ✓ generate select expression for timestamp s to
↪  s[2769601503-2769601503]
  ✓ generate select expression for timestamp s to s[-1429593275-None]
  ✓ generate select expression for timestamp s to
↪  s[3412549669-3412549669]

```

```
Add Custom Data Type In Runtime
  ✓ custom data type is added
  ✓ custom data type is applied[Some other string-Hello World]
  ✓ custom data type is applied[-None]
  ✓ custom data type is applied[None-None]
  ✓ custom data type is applied[ -Hello World]
  ✓ custom data type is applied[100-Hello World]
  ✓ custom data type is applied[0-None]
  ✓ multiple columns are accessed
  ✓ function name is shortened

unit/transformer/test_newest_by_group.py
Transformer to Group, Sort and Select the Top Row per Group
  ✓ logger should be accessible
  ✓ name is set
  ✓ str representation is correct

Transform Method
  ✓ Correct Row per Group (Single Column) is returned
  ✓ Correct Row per Group (Single Column) is returned
  ✓ Correct Row per Group (Single Column) is returned
  ✓ Correct Row per Group (Single Column) is returned
  ✓ Correct Row per Group (Single Column) is returned
  ✓ Correct Row per Group (Single Column) is returned
  ✓ Correct Row per Group (Multiple Columns) is returned
  ✓ Correct Row per Group (Multiple Columns) is returned
  ✓ Correct Row per Group (Multiple Columns) is returned
  ✓ Correct Row per Group (Multiple Columns) is returned
  ✓ Correct Row per Group (Multiple Columns) is returned
  ✓ Correct Row per Group (Multiple Columns) is returned
  ✓ Correct Row per Group (Multiple Columns) is returned
  ✓ Correct Row per Group (Multiple Columns) is returned
  ✓ Correct Row per Group (Multiple Columns) is returned
  ✓ Correct Row per Group (Multiple Columns) is returned
  ✓ Columns to Group by and Sort by are passed as Strings

unit/transformer/test_sieve.py
Mapper for filtering desired Elements
  ✓ logger should be accessible
  ✓ name is set
  ✓ str representation is correct

Filtering
  ✓ comparison
  ✓ regex

unit/transformer/test_threshold_cleaner.py
Cleaner based on ranges for numerical data
  ✓ logger
  ✓ name
  ✓ str representation

Cleaning
  ✓ numbers[integers]
  ✓ numbers[floats]
  ✓ non numbers

```

```
457
458 ========================= 361 passed in 153.52 seconds
    ↪  =========================
```