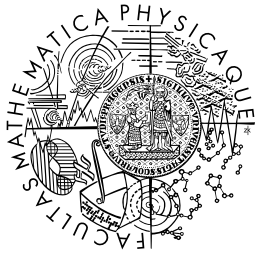

FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

MASTER THESIS

Andrej Jurčo

Data Lineage Analysis for PySpark and Python ORM Libraries

Department of Distributed and Dependable Systems

Supervisor of the bachelor thesis: doc. RNDr. Pavel Parízek, Ph.D.
Study programme: Computer Science
Study branch: Software and Data Engineering

Prague 2023

I hereby declare that I have authored this thesis independently, and that all sources used are declared in accordance with the "Metodický pokyn o etické přípravě vysokoškolských závěrečných prací".

I acknowledge that my thesis (work) is subject to the rights and obligations arising from Act No. 121/2000 Coll., on Copyright and Rights Related to Copyright and on Amendments to Certain Laws (the Copyright Act), as amended, (hereinafter as the "Copyright Act"), in particular § 35, and § 60 of the Copyright Act governing the school work.

With respect to the computer programs that are part of my thesis (work) and with respect to all documentation related to the computer programs ("software"), I hereby grant the so-called MIT License. The MIT License represents a license to use the software free of charge. I grant this license to every person interested in using the software. Each person is entitled to obtain a copy of the software (including the related documentation) without any limitation, and may, without limitation, use, copy, modify, merge, publish, distribute, sublicense and / or sell copies of the software, and allow any person to whom the software is further provided to exercise the aforementioned rights. Ways of using the software or the extent of this use are not limited in any way.

The person interested in using the software is obliged to attach the text of the license terms as follows:

Copyright (c) 2023 Andrej Jurčo
Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sub-license, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions: The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software. THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NON-INFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

In Prague date ....

Author's signature

This way I would like to thank to my supervisor, doc. RNDr. Pavel Parízek, Ph.D., for his help and advice whenever I needed anything.
Thanks should also go to RNDr. Lukáš Hermann for his valuable advice regarding MANTA Flow platform and making sure I had all the information I needed to work on the assignment.
I'm also extremely grateful to my parents, relatives and friends for their support throughout all my studies and without whom it would be very difficult do get this far.

Title: Data Lineage Analysis for PySpark and Python ORM Libraries

Author: Andrej Jurčo

Department: Department of Distributed and Dependable Systems

Supervisor: doc. RNDr. Pavel Parízek, Ph.D., Department of Distributed and Dependable Systems

Abstract: In the world of ETL tools and data processing, Python is one of the main languages used in practice. Python scripts that define data manipulations usually use the same Python framework, PySpark, which is the Python API for the Spark framework, alongside database libraries, using their ORM features. These ORM features usually work in a similar way in most of the relevant libraries. Recently, MANTA Flow, a highly automated data lineage analysis tool, was extended with a Python language scanner and now it is in the phase of being extended to support more commonly used frameworks.
In this work, we analyzed the PySpark library and the SQLAlchemy ORM technology in order to extend the MANTA's Python scanner with the support for these two frequently used tools. In case of the PySpark library, we designed and implemented a core of the plugin to the Python scanner which supports elementary functionality. The plugin is capable of analyzing various DataFrame input and output options available in PySpark for both file and database data sources, and it is able to propagate data flows during transformations with reasonable level of overapproximation, as demonstrated in the work. In case of the SQLAlchemy ORM, we designed a solution that would allow the scanner to analyze the ORM source code and its core could be used to support other libraries with ORM functionality as well.

Keywords: data lineage, data flow, python, symbolic analysis

Contents

1. Introduction

With the rise in the use of computers in every sphere of our lives, every business, government entity or citizen began to produce and collect a huge number of data. To put this huge amount into perspective, let's have a look at some facts. Between 1986 and 2007, digital storage grew 23% annually. By the time you finish reading this sentence, Google has probably processed over 100 thousand searches. Amazon stores approximately one exabyte of historical purchase data which is used for estimating future purchase needs [8].

With all this data over the years, everyone has been struggling with one problem: how to store data effectively? There have been many approaches to this problem and at first, they were probably sufficient. But it was most likely out of everybody's imagination how much it was going to grow in the recent years. And as a response, businesses started to make their already complicated data models even more complex. One extension after another, until they realized that their data storage is so complicated that they actually had no idea what is related to what. If you joined a bank in 70s and became a database architect, the chances are high that you are already retired and a huge portion of your knowledge left the institution with you. But the bank cannot just throw all your work away when you leave.

While there is no change and everything works, it is still alright. But then, data incidents happen. Invalid data in table XYZ. Nobody knows what is stored in table XYZ created 20 years ago, there is insufficient documentation and its name is also not very specific. The only way to find out is to have someone navigate the data ecosystem for a couple of days and then you figure out that the system that was producing data and stored them into the table had its firmware updated and used a different data format. Good news, you have found the problem and can fix this issue quite cheaply. All it cost you was a few days of work and an easy fix - this time you were lucky. But as the system grows, more and more incidents come across.

One day, your boss calls you and tells you that the predictions you made for the next fiscal quartal seem odd and asks you to perform a *data quality check*. But since you used seventeen tables in your prediction, you have to examine data origin and validity of all tables. In a large data environment, navigating to all sources of data for every table can take days, weeks, or even months. And you cannot simply skip it, because important business decisions are often based on data. If you have wrong business-impacting data, your decisions are most likely going to be wrong and that could have a serious impact on your business.

Same problems are dealt with by government entities. With each country having millions or tens-of-millions inhabitants, e-government solutions have to deal with huge amount of data. Data privacy must be ensured, too, to guarantee that unauthorized persons can not access your medical history or to make sure that your pension is not paid to someone else by accident through an error in data pipeline.

So what can be done to reduce data incidents, increase data transparency and improve data governance? To make navigation across the data solution (e.g. a data lake) easier, companies often decide to use data cataloging tools,

for example, Infogix's Data360, Informatica's Enterprise Data Catalog or IBM's Watson Knowledge Catalog. A data catalog tool automates the discovery of data sources throughout an enterprise's systems [25].

Having data catalogs certainly makes sense, but these tools do not tell you where did your data originate. For this, we need to create data lineage. Data lineage is generally defined as a kind of data life cycle that includes the data's origins and where it moves over time. It can help with efforts to analyze how information is used and to track key bits of information that serve a particular purpose [4]. Over the past few years, several companies managed to create their own data lineage solutions, one of which is the Czech-American company MANTA.

Starting as a project of the Czech software house Profinit , MANTA quickly became a separate company, selling its own product, MANTA Flow. MANTA Flow is a complex solution which is able to analyze data lineage in over 40 different technologies from the fields of data modeling and integration, business intelligence, databases or programming languages, one of them being the Python scanner, which is currently released as a prototype scanner. Covering a wide variety of technologies and having an ability to integrate well with data cataloging solutions, such as Collibra Data Governance Center or Alation Data Catalog, it gains a lot of popularity among the enterprise-level customers.

1.1 Goals

The goal of this thesis project was to extend the MANTA's Python scanner with additional functionality which would enable the scanner to analyze and find data flows in Python source code using features of the PySpark library and Object-relational mapping (ORM). Since the PySpark library is too complex to be supported completely, the scope of the support is reduced to the 'prototype' level - supporting elementary data-modifying and I/O operations is sufficient. Similar applies for ORM technology, where it is important to design such ORM support, that it is possible to reuse it for specific implementations for various libraries in the future.

The list of specific goals includes:

1. Analyze the Apache PySpark technology and understand the key concepts it uses.

2. Design and implement a new PySpark plugin for the Python scanner which will support elementary data-modifying and I/O operations.

3. Analyze the usage of ORM in Python and design a universal solution for its data lineage analysis in the Python scanner.

1.2 Glossary

Let us define some important terms that are often used in the whole text.

Data lineage is a representation of data relations between objects. It usually

maps the lifecycle of data, which means, that the reader of the representation should be able to find out where the data originates from, how it is used and modified during its life cycle and where it ends up. A visualization is usually a graph, but other representations can be used as well.

A *data flow* is a relation between two objects where one object provides the data and the other object consumes the provided data.

A *flow information*, in the context of symbolic analysis, is a location-specific representation of a piece of information obtained during the symbolic analysis of a programming language. It can be anything, like a string constant, a column of an SQL table which is loaded, or an item in a Python's `dict` object.

Static analysis is an automated analysis of source code which is performed without executing the code itself.

A *caller* is a language object (Python module, class or function) which invokes another object.

A *callee* is a language object (Python module, class or function) which is invoked by another object.

A *MANTA scanner* is a component of the MANTA Flow platform which is capable of performing data lineage analysis for a specific technology, such as a concrete DBMS, ETL tool, or a programming language.

A *scanner plugin* is a module of a programming language scanner which is responsible for a custom handling of some library functionality, without the simulation of the source code. This is common for specific functions interacting with other resources, like functions which perform file reads and writes, interactions with the database or printing to a console.

A *propagation mode* is a manual definition of the data flow handling of a certain function invocation. It specifies how function invocation input data flows shall be processed and *propagated* into the function invocation output data flows.

Deduction in the context of data lineage analysis is an information-inferring process from the surrounding code. It is used when the exact data model is not clearly defined in the source code. However, it is possible to get more information about the model indirectly, such as when pieces of the model are referenced in other operations. For example, when a specific column is queried over the result of a select query, it is reasonable to deduce that the selected column was a part of the result of the query.

1.3 Outline

At the beginning of this work, in Chapter 2, Data Lineage Analysis for Python, we briefly introduce MANTA Flow, the company's tool for data lineage analysis,

explain how it is extended to support more technologies, and discuss why is data lineage of programming languages important in the current world of data. In the end of the chapter, we explain how does the Python scanner work in a step-by-step manner, explaining what is happening and why it is necessary.

Chapter 3, named PySpark, introduces the PySpark technology, its relation to Apache Spark and describes its key modules. It then discusses how PySpark SQL module works - its data structures, actions and transformations, and metadata management in relation to data sources, sessions, and catalogs.

Chapter 4, Object–relational mapping[1], explains what ORM is, why it is useful and what are its advantages and disadvantages. Then, we proceed to describe how ORM works in practice, demonstrated on the SQLAlchemy ORM technology.

Chapter 5, Analysis, first defines the scope of the work based on the description of PySpark and ORM technologies, and then focuses on identifying all features necessary for supporting the mentioned technologies in the scope defined at the beginning of the chapter.

Chapter 6, Design, discusses the design of solutions to problems and features introduced in Chapter 5. It provides a detailed technical design that's implementation would result in the Python scanner's support for PySpark and SQLAlchemy ORM technologies in the defined scope.

Then, in Chapter 7, Implementation, we describe how individual features were implemented and highlight important implementation details worth mentioning.

Chapter 8, Evaluation, demonstrates the functionality of implemented features on several examples and discusses limitations of the implementation.

Lastly, Chapter 9, Conclusion, summarizes this work and compares the original goals of this work with the actual output.

There is also Section A, Attachments, at the end of this work which contains a short user documentation and describes the content of the attachment distributed with this work.

[1]Often abbreviated in this work as ORM.

2. Data Lineage Analysis for Python

Before we get to the main subject of this work, we first need to explain how MANTA works in general, why there even is a need to scan data lineage in Python source code and how it actually works.

2.1 MANTA Flow

As mentioned earlier, MANTA's flagship solution, named MANTA Flow, is a platform for automated data lineage analysis. It helps users to visualize their data pipeline, resulting in easier data governance, improved data quality and faster data incident resolution. The key feature of the solution is automation, thanks to which the solution from MANTA is able to perform the data lineage analysis in a couple of hours, or a couple of days for huge environments.

Because every technology stores data in a different way, having a different metadata structure, a new MANTA scanner must be produced for every supported technology. It is obvious that it is impossible to use the same approach for the extraction and the analysis of metadata from a DBMS, such as MS SQL or Oracle, and a BI tool, like PowerBI or Tableau.

Thanks to the well-designed MANTA Flow platform, the company is capable of abstracting the main concepts present in every technology and quickly develop new scanners according to the market needs - MANTA currently supports automated data lineage analysis for over 40 main technologies in the data pipeline - databases, data integration and modeling, or reporting and BI.

Every data lineage scanner has got the same high-level design - it consists of two main components, which are:

1. **Connector** which contains two main parts:
   (a) **Extractor** which extracts all inputs needed for the dataflow analysis from the server or other location, collecting all this information in a single location.
   (b) **Reader** which resolves scanner's inputs and creates their general model to be used for dataflow generation.
2. **Dataflow Generator** which analyses the reader output to create a data lineage graph which the user sees.

One might think that this is pretty much everything that is needed in the industry, but the fact is that to scan the whole customer environment, it is necessary to expand MANTA into new, and sometimes unknown areas, such as the analysis of programming languages.

2.2 Data Lineage in Programming Languages

It has become more common over time that technologies allow users to define their behavior by writing source code instead of using the user interface. For example, you can write procedures in C# in MS SQL, define JavaScript functions in Google BigQuery, or notebooks and jobs in Databricks can be written in Python, R, or Scala. This code, even though it is often very short, is very likely to create data flows which need to be analyzed and visualized together with data lineage of other technologies which use a much simpler approach.

2.2.1 Role of programming languages in data pipeline

There are many cases when MANTA Flow skips a part of data lineage analysis because it requires analysis of a source code and this may lead to certain inconsistencies in the resulting lineage graph. However, if the tool was able to allow other scanners to use the programming language scanners for the analysis of the source code embedded in other technologies, customers could be offered with a useful and unique feature that others could not deliver. Additionally, pieces of code do not have to be necessarily embedded in other technologies, they can be standalone programs executed as a cron job, or deployed on some third-party platform, like AWS Glue.

Static analysis of programming languages, which is also used in the Python scanner, is no easy job and even though there are several approaches which seem to work, there is no existing solution with the focus on data lineage. While the first phase of the static analysis, parsing (lexical and syntactic analysis), is relatively easy for programming languages due to a well-defined syntax, it is a major problem to perform the semantic analysis - understanding what the parsed code actually does and how it affects data. If we, for example, assign a value to a variable: `my_variable = 5+5`, it (usually) has got no impact for the data lineage and can be ignored - there is no data used from any data source (only numeric constants are used).

However, if we are loading some data from table A and writing this data to table B, we need to be able to recognize it and visualize it in the resulting data lineage graph by showing that data in table B originates in table A. For this and many other reasons, MANTA partners with Faculty of Mathematics and Physics of Charles University in Prague to conduct a research aiming to develop a working solution for static analysis of programming languages.

2.2.2 A happy day use case

To help you understand the context even better, let's see an example use case for the scanner and its value if everything works well, as illustrated in Figure 2.1. Imagine that you have a data pipeline consisting of four main components - a database and a transformation script which saves transformed data from the database into file(s). These files are then loaded into a BI tool, e.g. Qlik Sense or PowerBI.

Everything works perfectly until somebody in your large company makes some changes in the data transformation pipeline without letting anyone know and, suddenly, visualizations of data in your BI tool start showing weird data or stop

Figure 2.1: Without the Python scanner, you would not know how database and files are related (or you would not know there is any connection at all).

working completely. What would you do? If you combine tens of tables in your BI tool, finding the root of the problem can take several days. Luckily, you scanned your data pipeline using MANTA Flow before, and have the data lineage from the time when everything worked, as seen in Figure 2.2.

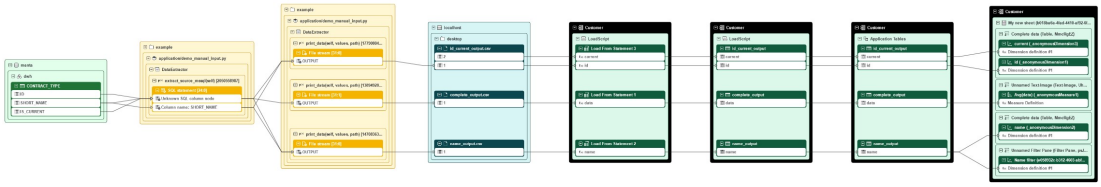

Figure 2.2: A simple data lineage visualization of 4 technologies (left to right: MS SQL, Python, Filesystem, Qlik Sense).

With MANTA Flow, all you need to do is run another scan of your pipeline and compare the before and after graphs (see Figure 2.3).

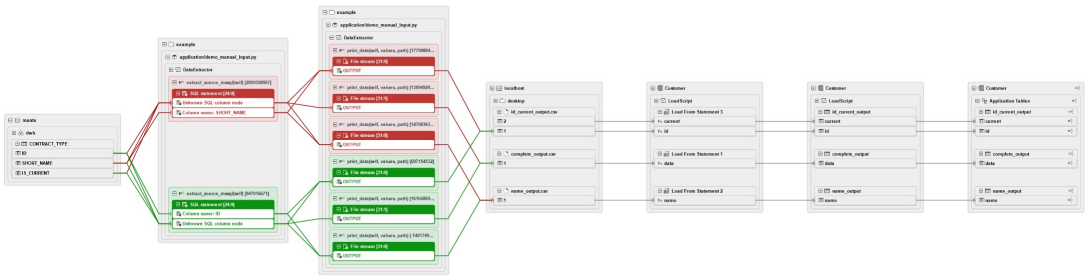

Figure 2.3: Comparison of two data lineage scans in MANTA. Red nodes are the nodes and edges which were removed, green are the new ones.

Luckily, you can see the problem easily - somebody changed the database columns which are stored in files and that is why your BI tool was suddenly making no sense. All you need to do in order to fix the problem is to get the problematic Python script back to its original state. Thanks to the fact that it was known how data in files is related to the database, it was possible to discover the issue in the matter of tens of minutes instead of a very long and tedious manual analysis of the whole pipeline [3].

2.2.3 Analyzing a programming language source code

It is very important to understand the whole process of the data lineage analysis for programming languages, how we start with just a couple of source code

files at the beginning and end up with their data lineage graph, before we can continue with the analysis and solution of more complex problems of this work. Let's briefly have a look at the workflow of data lineage analysis of a programming language. Excluding the COBOL scanner, which uses a different approach, MANTA currently develops three programming language scanners (for Java, C# and Python) which have similar workflow and we can abstract the way they all work. The sequence of individual steps can be seen in Figure 2.4.

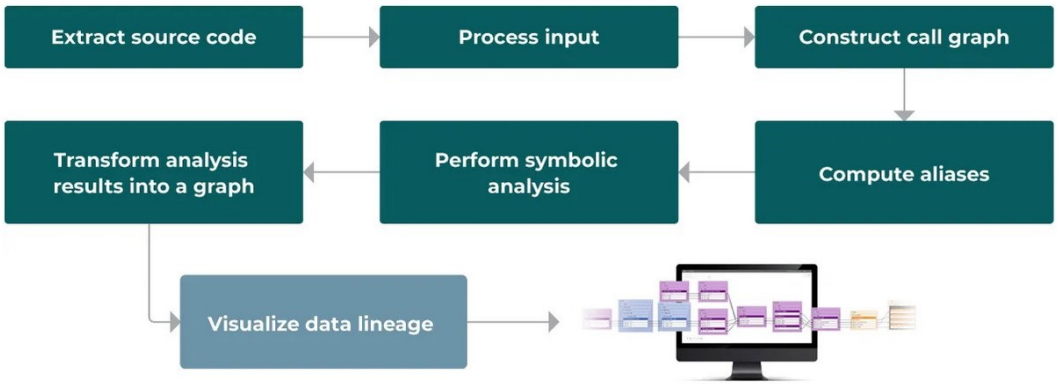

Figure 2.4: A simplified workflow of a programming language scanner.

Steps from the extraction of the source code until the analysis itself are performed in the *Connector* component, the creation of the resulting graph is done in the *Dataflow Generator*, which is shared among all three programming language scanners, and visualization is taken care of by the MANTA platform and its universal Visualizer component.

Extract the code

Before the analysis can begin, the scanner first needs to collect all input program code in some location. This step is called the extraction. Typically, two sets of input program codes are needed: application and libraries. *Application* is the source code provided by the user (the source code they have written and that's data lineage they want to see) and *libraries* is the source code used in the application, but the user did not write it, usually, these are publicly available libraries and Python built-in library.

Of course, the main focus is on finding out what the analyzed application does, and not really what does the implementation of libraries. However, it is essential to know which library functions are used - if the scanner sees that a `print()` function is called - is it the Python's native print into the console, or is the `print()` function of some other library performing a completely irrelevant action? For this reason, libraries have to be extracted and included in the analysis as well.

Process the input

Once all the input program code is collected, it needs to be loaded into the memory. There are countless ways to parse and process source code - if there is some Java code, it can be compiled and the output compiled code processed, the

source code can be parsed using own parser and grammar, or, in case of Python, an interpreter can be used to prepare a parse tree of the script and use that. For each programming language, the optimal solution is different and it needs to be chosen carefully to avoid as many potential problems as possible.

In case of the Python scanner, though, the path to parse the code on its own was chosen. The main reason behind this is the control of how the output would be represented, allowing a precise tailoring of representing data structures to the need of other components. This includes, for example, removal of syntactic sugar (instead of `'a' + 'b'`, an equivalent expression `'a'.__add__('b')` is used), simplifying expressions or adding implicit code constructs (instead of `'a' 'b'`, the scanner creates `'a'.__add__('b')`, adding the implicitly invoked concatenation function of string objects). On the other hand, one of the negatives is that the parser is not automatically compatible with all versions, but needs to be reviewed with every new version of the language, and a lot of source code which needs to be maintained.

Construct call graph

The next thing that needs to be done before the analysis can start is the computation of the call graph. The call graph captures dependencies between individual functions. If the analysis scope counts hundreds or thousands of files and functions, it is not always easy to determine which function is actually called. If we create a structure which can tell us quickly which are the functions that can be invoked in the current context, we can save a lot of computation time. This can be pre-computed at the beginning because function callers and callees do not change during the analysis.

The same stands for imported modules which contain functions that could be invoked and these imports do not change. Therefore, the component constructing the call graph must be able to resolve imports because it is very common that functions invoked are from various modules. If this information is lost, a gigantic amount of knowledge is gone.

Compute aliases

Step number four is to traverse the input program code and compute aliases. It is important to figure out which expressions reference the same data flow, allowing the scanner to correctly assign and, later during the symbolic analysis, to propagate data flows, even if we only one of the aliases is known. If we consider the following example:

```
a = 5   # a aliases value 5
b = a   # variable b aliases value 5 as well
```

From the analysis of the assignment statements it can be determined, without knowing the context, that when the variable `b` is used, it is the same as using the variable `a`. A traversal of the assignment statements in the input program code representation can help with computing aliases per different parts of code - functions, classes, or modules. Again, aliases are context-insensitive (therefore, the exact variable value is not stored, it is just noted that `a` is the same thing as `b` and this information can be used for the given part of code.

Run the analysis

When all context-insensitive tasks are done, the actual symbolic analysis can commence. Performing symbolic analysis means that the input source code is analyzed, without actually being run. However, it loses some precision in cases when a runtime value is provided. A good example are control statements. Let's have this if-statement:

```
option = input('Choose output destination: ')
if option == 'file':
    # write data into a file
else:
    # write data into the database
```

If the program was analyzed at runtime, it would be known what is the value of the option variable. However, in static analysis, it can only be guessed. To avoid missing the correct option, it must be assumed that both options are correct. Thus splitting the execution into two branches - one branch counts with the fact that the `option == 'file'` condition was fulfilled, and the other one pretends that it was not.

Typically, an execution of a program has some entry point - a function or a file which is executed (for example, the `main()` function in Java, or the Python module which is ran by the interpreter). This is where the execution starts and whence all invocations, branching and operations need to be computed.

During the analysis, all objects (in case of Python, these are modules, classes, and functions) which need to be analyzed are placed in a worklist (an enhanced queue). Objects in the worklist are those which are reachable from the entry point (by invocation in the source code). The analysis uses *Invocation context* which is the context of the program at the point in the source code where the function was invoked. Because contexts can differ, for example due to branching at conditionals and other language constructs, the analysis must compute *flow summary* for every context. The flow summary of an executable (module, class, function) is the flow information computed over the object across all contexts. If a flow summary for an executable changes, all its callers and callees must be added into the worklist again, because their flow summaries may be affected.

The analysis runs in a loop while the worklist is not empty. At that moment, the analysis ends because there is nothing new to be computed anymore and the results are transformed to be used by the *Dataflow Generator*.

An obvious question you might be asking now is - the number of options and possible execution branches is huge, can the analysis ever finish? Well, yes and no. In order to get a relatively accurate output in a reasonable time, some approximations have to be used which would not reduce the correctness of the analysis - for example, the scanner does not simulate the execution of library function, but instead, it contains *propagation modes* for the important library functions which handle data flows in a special way (specific for every function), without a need to simulate their body.

If these modifications are implemented correctly, the analysis finishes in a relatively short time with a reasonable overapproximation of the resulting graph (that is, there are some false positives, e.g. some incorrect data flows, but all actual flows are present). However, only explicitly specified flows can be tracked and

what is not in the user source code, or specifically implemented in a propagation mode, as if did not exist at all.

Create the resulting graph

The last step before a data lineage graph is ready to be visualized is, as mentioned earlier, to transform the output of the static analysis into a standard MANTA graph. During the analysis itself, it makes no sense to represent the data as graph because different contexts produce different nodes and edges which sometimes turn out to be worthless in terms of data lineage.

Instead of adding nodes and later removing them, all context data is kept in separate data structures which are easy to work with and are only transformed into the graph at the end, adding nodes and edges only for those data flows which are interesting for the user (file and database streams, console input/output etc.). Creation of the graph is done via the common *Dataflow Generator* for all three programming language scanners because the output of the *Connector* can be abstracted into a form which does not require any language-specific logic anymore.

Visualize data lineage

Allowing the user to see data lineage is taken care of by the MANTA Visualizer. It is a component which visualizes graphs from different scanners and creates a complete picture of the data lineage for the user.

When compared to MANTA graphs of other scanners, it is clear that programming language graphs are way smaller. The reason behind this is very simple - if every step of the program was visualized, with all the branching and all context options, there would be an incomprehensible graph in the end which would provide no value to users. Users would only be capable of reading it if they contracted inner nodes and saw a smaller graph. This raises a question - is the big graph even needed, or is it enough to only show those points in the program which interact with external resources, as shown in Figure 2.5?

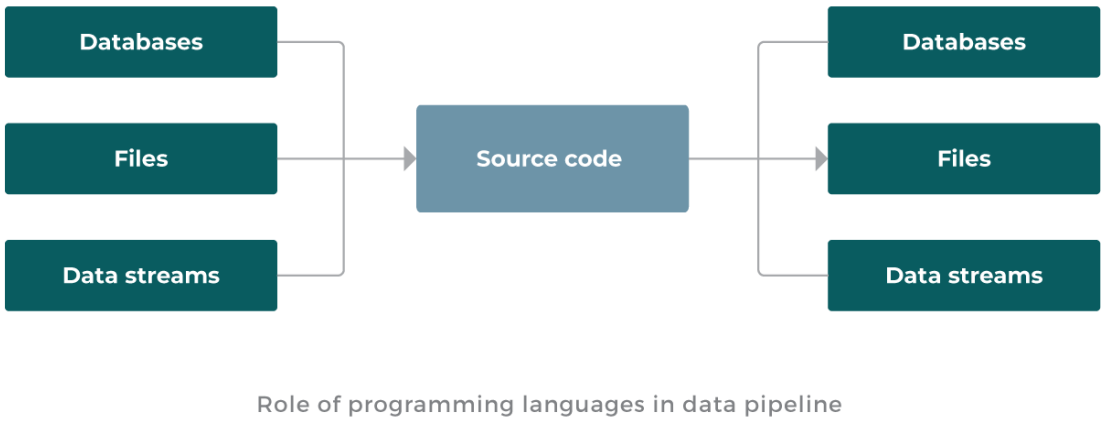

Figure 2.5: Relevant parts of a programming language data lineage graph.

More compact graphs make sense - only the ‘entry points’ flowing into/from the given program point are visualized. For a user, it is not very useful to see

that you initialize a variable with a value 5 on line 427, if the variable plays no role in terms of data lineage. What is more valuable in the world of data lineage is seeing that the Python program loads data from table `xyz`, processes it inside (without visualizing exactly how), and writes the output into a CSV file - the user knows that the source of the CSV file's data is in the table `xyz`, perhaps with some modifications. If the CSV file is then used to load data into a BI tool, the user will see the relation between the BI tool data and the table `xyz` which would otherwise be unknown had the Python scanner not been used.

Below, in Figure 2.6, you can find an example of how the data lineage looks if the data is loaded from a database and printed to a console. Note that in some cases, the exact column name or index are unknown during the analysis since the code is not run — in such cases, it may need to be claimed that there are some columns used, but the scanner was unable to enumerate them all (only the program code is analyzed, not SQL queries or content of read/written files). The green node is the output of the MANTA's MS SQL scanner and yellow nodes are from the Python scanner.

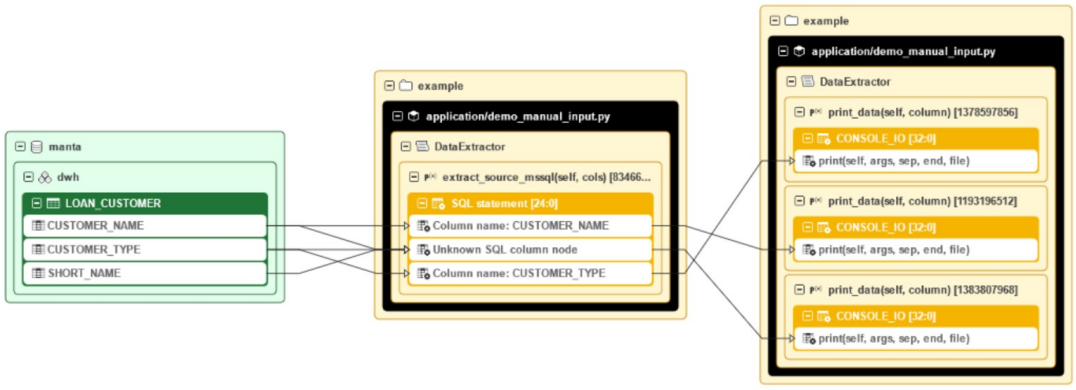

Figure 2.6: A visualization of a Python program data flows loading data from a database and printing different subsets of the SQL query result to the console.

3. PySpark

In this chapter, we are going to introduce the PySpark library, its features and key concepts. Because the library is of significant size, we are going to focus on those parts of the library which are relevant to the rest of this work.

Thanks to Python's easy-to-use syntax and ability to write simple programs in a matter of minutes and many big data libraries available, the language is often a go-to option when data scientists or developers need to write scripts for processing data. According to the JetBrains Python Developers Survey 2021 [18], more than half of Python developers use it for Data Analysis, making it its number one purpose. Indeed, when we have a look at all available libraries, we can find libraries for:

- computation in distributed systems, e.g. PySpark,
- simple data manipulation and representation, e.g. Pandas,
- representing complex mathematical structures, e.g. NumPy,
- plotting, e.g. matplotlib, or,
- machine learning and artificial intelligence, e.g. TensorFlow.

In the world of big data, the number one Python library is PySpark [18]. Approximately one in nine developers uses this library, and, when combined with other Apache's products, such as Kafka, Hadoop, and Hive, they dominate the sector.

When we compare PySpark to other data science libraries, we can observe that the ability to easily utilize distributed computation distinguishes it from its competitors, such as Pandas.

3.1 Overview

PySpark is a Python interface for Apache Spark. It not only allows users to write Spark applications using Python APIs, but also provides the PySpark shell for interactively analyzing data in a distributed environment. PySpark supports most of Spark's features such as Spark SQL, DataFrame API, Streaming, MLlib (Machine Learning) and Spark Core [17], as illustrated in Figure 3.1.

At a high level, every Spark application consists of a driver program that runs the user's main function and executes various parallel operations on a cluster. The main abstraction that Spark provides is a resilient distributed dataset (RDD), which is a collection of elements partitioned across the nodes of the cluster that can be operated on in parallel. RDDs are created by starting with a file in the Hadoop file system (or any other Hadoop-supported file system), or an existing Scala collection in the driver program, and transforming it. Users may also ask Spark to persist an RDD in memory, allowing it to be reused efficiently across parallel operations. Finally, RDDs automatically recover from node failures [19].

The main use case for PySpark is, as described above, to perform high-performance computing over various datasets. The secret ingredient in PySpark's

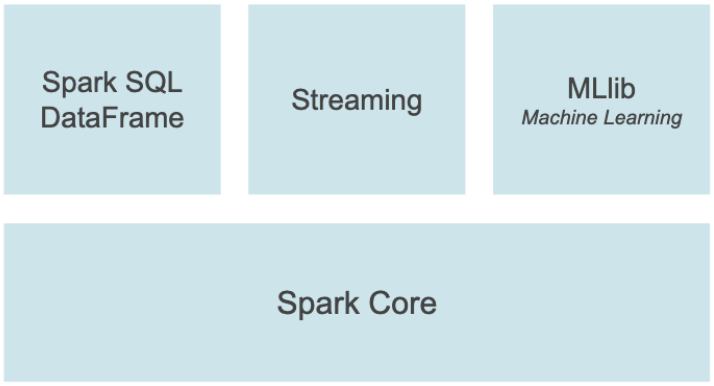

Figure 3.1: Overview of the Spark architecture. Source: Apache PySpark documentation.

favor is the fact that the data structures used for representing data in (Py)Spark can be distributed across several machines and the system can, therefore, process even the largest tables easily. Distributed computing is done on the background, which means, that after some setup, users do not have to deal with problems of distributed computing by themselves at all.

3.2 Spark Streaming

Spark Streaming extends the core Spark API with capabilities of a scalable, high-throughput, fault-tolerant stream processing of live data streams. It allows loading data from various sources, such as Kafka, Kinesis, or TCP sockets. Additionally, it can process the data using complex algorithms expressed with high-level functions like map, reduce, join and window. The processed data can be stored into filesystems, databases, and live dashboards [22]. The workflow can be seen in Figure 3.2.

Figure 3.2: Visualization of the Spark Streaming workflow. Source: Apache Spark documentation.

We can demonstrate how it can be used on a very simple use case from the Spark's documentation [22], as shown in Figure 3.3.

```
1  from pyspark import SparkContext
2  from pyspark.streaming import StreamingContext
3
4  sc = SparkContext("local[2]", "NetworkWordCount")
5  ssc = StreamingContext(sc, 1)
6  lines = ssc.socketTextStream("localhost", 9999)
7
8  words = lines.flatMap(lambda line: line.split(" "))
9  pairs = words.map(lambda word: (word, 1))
10 wordCounts = pairs.reduceByKey(lambda x, y: x + y)
11
12 wordCounts.pprint()
13
14 ssc.start()
15 ssc.awaitTermination()
```

Figure 3.3: A simple program using the Spark streaming functionality. Copied from PySpark documentation [22].

In this example, a new context with two execution threads is created (line 4), then, the batch interval is set to one second (line 5). After the setup, a data stream from a TCP source at `localhost:9999` is created. This source is first used to split the data by a whitespace character into words and then to count each work in each batch (lines 8 and 9). Finally, a reduction is used to sum up the word frequency occurrence and print it into the console (lines 10 and 12). Once this pipeline is set up, the program can be started and let to run until it is terminated (lines 14 and 15). In a couple of lines, it is possible to create a simple PySpark streaming application.

3.3 Spark MLlib

MLlib is a scalable Spark library used for machine learning. It provides a large amount of uniform high-level APIs which make creating machine learning pipelines easy and intuitive. Overall, it provides several API sets:

- ML Algorithms,
- featurization,
- pipelines,
- persistence,
- utilities [9].

Thanks to this library, it is very easy to perform the majority of machine learning operations. For example, to create a simple summarizer, only a single line of code is needed, as shown in Figure 3.4.

```
1  from pyspark.ml.stat import Summarizer
2  from pyspark.sql import Row
3  from pyspark.ml.linalg import Vectors
4
5  df = sc.parallelize([
6       Row(weight=1.0, features=Vectors.dense(1.0, 1.0, 1.0)),
7       Row(weight=0.0, features=Vectors.dense(1.0, 2.0, 3.0))])
8     .toDF()
9
10 # create summarizer for multiple metrics "mean" and "count"
11 s = Summarizer.metrics("mean", "count")
12
13 # compute statistics for multiple metrics without weight
14 df.select(s.summary(df.features)).show(truncate=False)
15
16 # compute statistics for single metric "mean" without weight
17 df.select(s.mean(df.features)).show(truncate=False)
```

Figure 3.4: A simple program using the Spark MLlib module functionality. Copied from PySpark documentation [23].

3.4 Spark SQL

Spark SQL is a Spark module for structured data processing. When compared to Spark RDD API, the main difference is that the interfaces provided by Spark SQL allow for more structure of both the data and the computation being performed. Spark SQL uses the extra information about the structure to perform additional optimizations. Commonly, Spark SQL offers two ways of usage - Spark SQL including SQL and the Dataset API. It uses the same execution engine for both approaches and regardless of the language used (Scala, Java, Python, and R). This allows users to choose the best way to express the transformation to be done, setting very few constraints [21].

3.4.1 Datasets and DataFrames

A Dataset is a distributed collection of data which was added in Spark 1.6, released in November 2016. It provides the benefits of RDDs (strong typing, ability to use powerful lambda functions) with the benefits of Spark SQL's optimized execution engine. Datasets are constructed from JVM objects and then manipulated using functional transformations (map, flatMap, filter, etc.). This API is available in Scala and Java, Python does not have the support for it. Due to Python's dynamic nature, many of the benefits of the Dataset API are already available (i.e. it is possible to access the field of a row by name naturally `row.columnName`) [21].

On the other hand, a DataFrame is a Dataset organized into named columns. It is conceptually equivalent to a table in a relational database or a data frame in R/Python, but with more optimizations. DataFrames can be constructed from a wide array of sources such as: structured data files, tables in Hive, external

databases, or existing RDDs. The DataFrame API is available in Scala, Java, Python, and R. In Scala and Java, a DataFrame is represented by a Dataset of Rows [21].

In the rest of this work, we are going to focus on DataFrame only as it makes no sense to consider Datasets since they are not present in PySpark.

To allow users to specify column names and data types, PySpark supports definition of DataFrame schemas. A DataFrame schema is a collection of schema fields (class `StructField`) organized in a wrapper class, `StructType`.

The `StructField` is a container for the column name, data type (such as a string, an integer, or a date), *nullable* flag and, optionally, column metadata.

On the other hand, the wrapping class `StructType` only provides additional functionality for adding new columns, loading structure from a JSON object or retrieving a list of all contained column names.

If a user wants to create a new DataFrame, they can do it in several ways. Most commonly, a DataFrame from a data source is used, such as from a CSV or a parquet file, or a database table (see Section 3.4.3 for details). A user may, additionally, omit the schema completely and let PySpark infer it, including column data types. Loaded data is then processed to fit the schema correctly, however, if the resource does not contain a header, column names are generated by combining an underscore prefix with the column position, starting at 1. Therefore, three columns with generated names would, for example, be `_1`, `_2`, and `_3`. Another way to create a DataFrame is to initialize it by utilizing the SparkSchema's `createDataFrame(...)` function. It creates an empty DataFrame with the schema provided as one of the parameters. If no schema is provided, the new DataFrame has no columns.

An interesting feature of DataFrame is the ability to access its columns by *attribute access*. If there is, for example, a column `A` in a DataFrame `df`, one may simply use command `df.A` to get access to the column. This is possible thanks to the implementation of the `__getattr__()` function in the DataFrame class.

Every initialized DataFrame has got a context in which it operates, represented by the *SparkSession* object. Both data source reading functions and `SparkSession.createDataFrame(...)` initialization options clearly define the *SparkSession* instance which is the parent session of the DataFrame. We will talk more about sessions in Section 3.4.4 - at the moment, we don't need more details.

Another very important feature of DataFrame is its immutability. If a user decides to do any transformation with a DataFrame, the target object never changes. Instead, every transformation creates a new DataFrame instance that contains data *after* the transformation.

3.4.2 Transformations and Actions

Everything that a user can do in the PySpark's SQL module, which is going to be the main focus for us, can be split into two categories:

- *Transformations* - when a transformation is applied, the result of the computation produces a new data structure with the transformation applied to all records. A new object is created because these data structures in PySpark SQL are immutable and, therefore, no transformation can be applied on them (they are read-only). An example of such data transformation can

be functions `map()` or `filter()`, while structure transformations are, e.g., functions `drop()` or `select()`, which modify the structure of a DataFrame.

- *Actions* - no new object is created, the main use case for this type is retrieving a single value or persisting of data, for example, functions `count()`, `reduce()` or `saveAsTextFile()`.

Before we have a look at the way I/O operations work in PySpark, let's have a quick look at some elementary and common transformations supported by DataFrames. We will omit too much detail and implementation specifics, so that the reader does not get too distracted with non-essential information. Most information in the overview below is from the PySpark Documentation itself [17].

- `alias(alias)` - adds a DataFrame alias, creating a new column alias with the DataFrame alias prefix.

  ```
  df_aliased = df.alias("df_as1")
  df_aliased.select('df_as1.name', 'df_as1.age')
  ```

- `crossJoin(other)` - performs a cartesian product with another DataFrame. The resulting DataFrame's *SparkSession* is that of the target DataFrame, not the 'other'.

- `drop(*cols)` - drops the specified column (or more columns), the parameter is either a string name of the column or the column instance to be dropped. If no matching column is found in the target DataFrame, no operation is performed.

- `join(other, on=None, how=None)` - performs a join operation, using the expression provided. If not present, inner join is performed.

- `select(*cols)` - projects `*cols` provided and returns a new DataFrame representing the result of projection. Columns can be either strings (names of columns to be selected), instances of the `pyspark.sql.Column` class, or lists of strings or `Columns`.

- `union(other)` - performs an equivalent to SQL's UNION ALL over two DataFrames - `self` and `other`. The `self` DataFrame is the base for the new resulting DataFrame in terms of object properties. Union is performed by position, not by a name, as it is done in SQL as well.

- `unionAll(other)` - equivalent to `DataFrame#union`.

- `withColumn(colName, col)` - creates a new DataFrame instance from the `self` DataFrame by adding the `col` column under the name `colName`. If a column with such name already exists, it is replaced.

- `withColumnRenamed(existing, new)` - returns a new DataFrame which was created from `self` by renaming the column with name `existing` to `new`. If there is no column with `existing` name, no operation is performed.

A very simple program which uses some of the above-listed functions can be seen in Figure 3.5. Note that this program is only for the demonstration and as such does not really work with real data and does not perform any useful task.

```
from pyspark.conf import SparkConf
from pyspark.sql.session import SparkSession
from pyspark.sql.types import StructType, StructField, \
                              IntegerType, StringType
from pyspark.sql.functions import lit

# create a schema
f1 = StructField("id", IntegerType(), False)
f2 = StructField("txt", StringType(), False)
my_list = [f1, f2]

# initialize sessions
conf = SparkConf()
spark = SparkSession.builder.getOrCreate()

# provide data and DataFrame schema
df2 = spark.createDataFrame(
    [(1, "foo"), (2, "bar"),],
    StructType(my_list),
)

df = spark.read.csv("./input.csv")
# transformations always create a new DataFrame
df = df.alias('df_alias')
df = df.crossJoin(df)
df = df.withColumn('column1', lit(None))
df = df.withColumn('column2', lit(1))
df = df.withColumnRenamed('column2', 'col')
df = df.unionAll(df2)
df = df.drop('column1')

# df: DataFrame[col: string (+ content of input.csv)]
# df2: DataFrame[id: int, txt: string]
print(df2)
df.write.csv("my_file.csv")
```

Figure 3.5: An example PySpark program performing transformations.

3.4.3 PySpark I/O operations

After getting familiar with PySpark's DataFrames and the main functionality that we are going to focus on in terms of transformations, we can have a look at how DataFrames can be loaded from and written to data sources. For the sake of simplicity, we are going to ignore the functionality which allows defining own loading/writing mechanisms through configuration of the Spark engine, and will focus on formats and functionality supported in PySpark by default.

For I/O operations, PySpark uses two objects - `DataFrameReader` for reading data sources and then creating DataFrames, and `DataFrameWriter` for writing DataFrames into data sources. These objects support reading or writing over several different data source formats, namely CSV, JSON, PARQUET, ORC, and text files, and databases via JDBC connections. While for files, the resolution of path and options is rather straigh-forward, there is a larger context behind resolving JDBC connections and processing SQL. Therefore, we discuss the relation between PySpark and SQL in a greater detail in Section 3.4.4.

`DataFrameReader` and `DataFrameWriter` class instances, in this section referenced as *reader* and *writer*, are obtained slightly differently.

For reader, a `SparkSession` instance must be used, which utilizes the `@property` decorator to initialize the `read` attribute:

```
@property
def read(self):
    return DataFrameReader(self._wrapped)
```

Then, whenever a reader needs to be retrieved for the given session, let's say it is stored in the variable `session`, only a simple `read` attribute access is necessary:

```
reader = session.read
```

Every access to the `read` attribute creates a new reader object which means that configuration of a reader is not shared among two attribute accesses.

```
reader1 = session.read
# configure reader1
reader2 = session.read
# reader2 is configured to default values, regardless of reader1
```

A similar approach is used for obtaining the writer. The only difference is that the `write` attribute is accessed on the DataFrame instance, not a `SparkSession`. The behavior is, then, the same as for the reader - a new instance of the writer is returned upon every `write` attribute access.

This behavior, when a library object's implicit field is accessed, is a potential problem for the scanner, because initialization of attributes `read` and `write` is nowhere in the code and, therefore, when accessed, the Python scanner has no idea what this attribute is or what should be written. We discuss more on this topic later in Section 5.3.

For every standardly supported format, the reader and the writer contain a specific function to be used, such as `json()` or `jdbc()`. These functions are heavily parameterized, mostly for specifying formatting options, such as `timestampFormat` or `ignoreNullFields`. As a result, functions can have as

many as thirty-five parameters in case of the `DataFrameReader#csv` function. Almost all of these parameters are optional, though.

There are, however, also generic functions for loading and writing DataFrames. In fact, format-specific functions call generic functions with only setting some options implicitly, such as `format="csv"` for the `csv()` function. Functions `load()` and `save()` aggregate all options available for the given reader or writer object and then, based on these options, perform the operation. Not all options have to be set explicitly, though. For example, the default value for the format of a data source is `parquet`.

In addition to all these I/O options, there is more functionality available. The reader provides the `table()` function, which returns a DataFrame representing the specified database table. Similarly, writer's function `saveAsTable()` saves a DataFrame into the specified table. To determine behavior on-write, `mode()` function is available with four options:

- *append*: Append contents of this DataFrame to existing data.
- *overwrite*: Overwrite existing data.
- *error or errorifexists*: Throw an exception if data already exists.
- *ignore*: Silently ignore this operation if data already exists [17].

It is also important to know how options can be specified. Alongside passing options as parameters when invoking I/O functions, there are also two option-setting funcitons available for both reader and writer - `option()` and `options()`.

The `option(key, value)` function simply stores the provided `value` in the reader's or writer's dictionary of options under the `key` key.

Similarly, the `options(**options)` function goes over all `key=value` pairs and stores them in the options dictionary.

It is important to mention that option functions return the reader or writer instance, making it possible to chain invocations. Therefore, loading a new DataFrame can look as follows:

```
session = SparkSession.builder.getOrCreate()
jdbc_df = session.read \
    .format("jdbc") \
    .schema(StructType([StructField("col1", IntegerType())])) \
    .option("url", "jdbc:oracle:thin:@localhost:1521/orcl") \
    .option("dbtable", "schema.tablename") \
    .option("user", "username") \
    .option("password", "password") \
    .load()
```

Similarly, for writing a DataFrame, the code can look like this:

```
jdbc_df.write \
    .format("csv") \
    .path("my_file.csv") \
    .options(mode="append", sep=";")  \
    .write()
```

3.4.4 Spark SQL core data structures

In PySpark, there are several abstractions which help developers and data analysts with creating complex, but relatively simply understandable, data pipelines, mostly relying on databases. This is the main use case of PySpark SQL, an API giving access to Spark SQL core functionality, which has already been briefly summarized in this section. Now, we are going to describe how the whole Spark SQL "core" works on the background. This applies to all APIs and implementations of Spark - Spark (Scala, Java), PySpark (Python), and SparkR (R language).

When working with Spark SQL, the two elementary data structures responsible for managing the whole context of Spark runtime are classes `SparkContext` and `SparkSession`.

SparkContext

The `SparkContext` class is the entry point for Spark functionality. It represents the connection to a Spark cluster and can be used to create RDDs, accumulators and broadcast variables on that cluster. Only one SparkContext should be active per JVM [17].

SparkSession

`SparkSession`, on the other hand, is the entry point to programming Spark using Dataset and DataFrame APIs (in case of PySpark, it recognizes only DataFrame, Dataset API functionality is included in the DataFrame API). SparkSession is present since Spark 2.0.0. Before that, SQLContext was used to work with rows and columns (Spark 1.x) [17]. Spark 2.0 was released already in July 2016 and, therefore, we are going to ignore Spark 1.x, for the sake of simplicity.

It is worth noting that there can be several SparkSessions created, with each instance having its own `SessionState`. If a session has been created upfront, an instance can be obtained through the `SparkSession.Builder` object using the command `SparkSession.builder.getOrCreate()`. To create a new session with custom configuration, following commands can be used:

```
SparkSession.builder
     .master("local")
     .appName("Word Count")
     .config("spark.some.config.option", "some-value")
     .getOrCreate()
```

The `SparkSession.Builder` lookup algorithm works as follows:

1. Check whether there is a valid thread-local SparkSession, and if yes, return that one.
2. Check whether there is a valid global default SparkSession, and if yes, return that one.
3. Create a new SparkSession and assign the newly created SparkSession as the global default.

4. In case an existing SparkSession is returned, the non-static config options specified in this builder will be applied to the existing SparkSession [17].

SparkSession can also be created in different ways, for example, using the `SparkSession.newSession()` function. However, using Scala's *object* functionality, there is also a 'global' instance of the session which provides additional mapping and instantiating options, i.e., provides already created instances of a session when parameters match the builder and method `getOrCreate()` is used.

3.4.5 Session and Shared States

When working with sessions, there are two different states that can be used: the global `SharedState` and the instance-specific `SessionState`. The SharedState is a singleton (again, via Scala's *object* concept) and, as the name suggests, can be accessed by any SparkSession, as opposed to the SessionState, which is private for every session instance.

Every SessionState provides, as you can see in Figure 3.6, among other details, information about tables, views, columns etc., available in the local (session) scope. This information is stored in a `SessionCatalog` instance, which keeps all information about available objects. However, access functionality is mediated by the Catalog object, which is a direct attribute of the session instance. If it wasn't like that, all operations would have to be invoked using `SparkSession.sessionState.<function>(args)`, which is not very convenient.

Therefore, every `SessionCatalog` instance stores various table and database information and, in addition, allows users to store views. A view is, for example, the result of persisting some DataFrame data. When data is loaded from a file into a DataFrame instance, it can be stored in a SessionCatalog as a local view, or it can be stored globally, so that every session can access it.

Moving forward to the `SharedState` class, it can be said that its instance provides two simple functionalities:

1. Creation and maintenance of a `GlobalTempViewManager` instance, so that views can be accessed from any session, which can be useful when a user wants to prepare common views at the beginning, right after JVM starts, and all their sessions do something different, but over the same set of views.

2. Initialization of the `ExternalCatalog` class, which provides available databases, tables, or columns initialized from the linked *Apache Hive* server or other external data store. The ExternalCatalog is just a *trait* and within the Spark library, it is only implemented for the usage with the Apache Hive technology. Other implementations may be supplied, but they are not provided as part of the standard Spark library.

3.4.6 Tables and Views

As described earlier, SparkSession keeps its shared and session states, which manage, among other objects, also loaded and/or created tables and views. In PySpark, views and tables are stored using functions listed below.

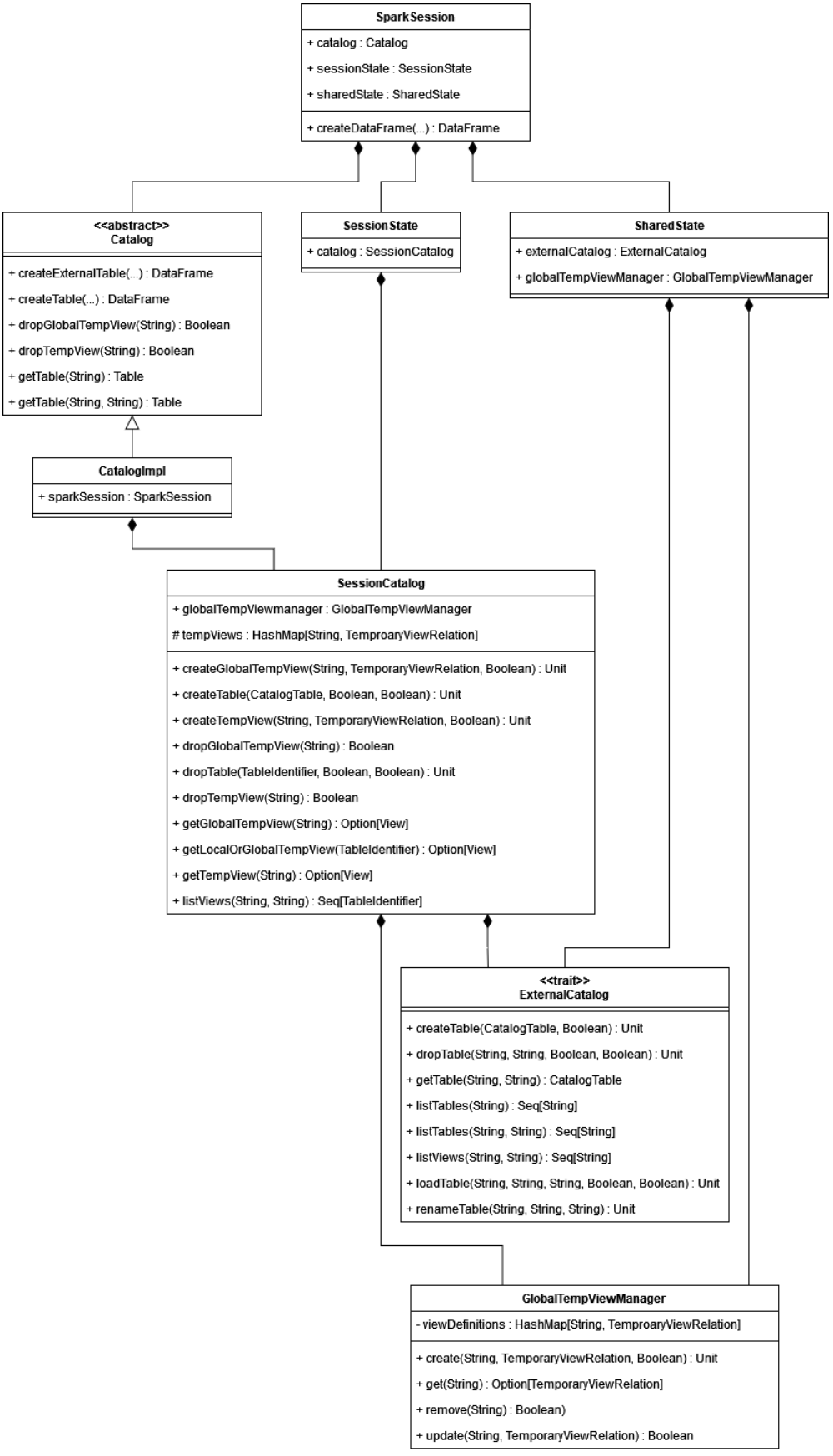

Figure 3.6: Simplified relations of SparkSession-related objects in Spark SQL.

- `DataFrame.createTempView(self, name) → None`
  Saves the target DataFrame as a temporary view in the session state under the defined name.
- `DataFrame.createOrReplaceTempView(self, name) → None`
  Saves the target DataFrame as a temporary view in the session state under the defined name. If this view already exists, it is overwritten.
- `DataFrame.createGlobalTempView(self, name) → None`
  Saves the target DataFrame as a temporary *global* view in the shared state under the defined name.
- `DataFerame.createOrReplaceGlobalTempView(self, name) → None`
  Saves the target DataFrame as a temporary *global* view in the shared state under the defined name. If this view already exists, it is overwritten.
- `DataFrame.registerTempTable(self, name) → None`
  Registers a DataFrame as a temporary table using the given name. The lifetime of this temporary table is tied to the SparkSession that was used to create the DataFrame. Deprecated as of PySpark 2.0.0.

Cross-session operations

When there are operations over several DataFrame instances, such as `join()` or `union()`, where every DataFrame instance may have a different parent SparkSession, the resulting DataFrame's parent session is the one of the invocation target object ("`self`").

```
spark = SparkSession.builder.getOrCreate()    # session instance 1
spark2 = spark.newSession()                   # session instance 2

df1 = spark.createDataFrame([], StructType([])) # parent session 1
df2 = spark2.createDataFrame([], StructType([]))# parent session 2

df3 = df1.join(df2)                             # parent session 1
df4 = df2.join(df1)                             # parent session 2
```

3.5 Relation to the pandas library

Finally, it is worth noting that PySpark is closely related to the pandas library - their data structures representing tables and data are transformable from/to each other. To access pandas functionality on a PySpark DataFrame, function `DataFrame#to_pandas` has to be used.

```
import pyspark.pandas as ps

pyspark_dataframe = ps.range(10)
pandas_dataframe = pyspark_dataframe.to_pandas()
```

To convert a pandas DataFrame instance to its equivalent PySpark DataFrame representation, the `SparkSession#createDataFrame` function can be used.

```
import pandas as pd
from pyspark.sql import SparkSession

data = [['Scott', 50], ['Jeff', 45]]
pandasDF = pd.DataFrame(data, columns = ['Name', 'Age'])
spark = SparkSession.builder.getOrCreate()
spark_df = spark.createDataFrame(pandasDF)
```

4. Object–relational mapping

In this chapter, we are going to introduce the ORM technology and its key features. It is important for the Python scanner to be able to analyze the source code which uses this technology as it produces a lot of data lineage which is very important for users. The technology allows users to use databases in Python script in a very different manner when compared to the usage of common database libraries and supporting this feature allows the scanner to be used in a large amount of new use cases.

Before we get to technical details of this work, let us first introduce the concept of Object-relational mapping (from now on, abbreviated as *ORM*), its advantages, disadvantages, and its common use cases. It is necessary to understand how this technology works before we can start the analysis and design to support ORM libraries in the Python scanner.

Note that the majority of code snippets in this chapter are taken from the SQLAlchemy ORM documentation [24] with occasional minor adjustments to highlight the subject described in individual sections.

4.1 What is ORM?

Object-relational mapping is a programming technique which allows developers to map data present in relational databases to object code, using metadata descriptors. The object code is written in one of the object-oriented programming languages, such as Python, Java, or C# [10].

It allows developers to work with objects in their applications instead of relying on embedded SQL code, which allows for faster development and better readability of the code.

If a developer wants to use ORM in their code, its configuration is fairly simple.

Firstly, the developer needs to specify the model to which data shall be mapped. This is usually done by declaring classes which serve as data structures representing individual (sets of) objects. To declare two mapped classes, `User` and `Address`, with a foreign key from Address to able User, we can do so, for example, followingly:

```
class Base(DeclarativeBase):
    pass


class User(Base):
    __tablename__ = "user_account"

    id: Mapped[int] = mapped_column(primary_key=True)
    name: Mapped[str] = mapped_column(String(30))
    fullname: Mapped[Optional[str]]

    addresses: Mapped[List["Address"]] = \
```

```
                    relationship(back_populates="user")


class Address(Base):
    __tablename__ = "address"

    id: Mapped[int] = mapped_column(primary_key=True)
    email_address: Mapped[str]
    user_id = mapped_column(ForeignKey("user_account.id"))

    user: Mapped[User] = relationship(back_populates="addresses")
```

Then, it is important to configure the library or the system providing the ORM functionality, so that it is clear with which data source the ORM system works:

```
engine = create_engine('mssql://user:pass@localhost:1433/some_db')
session = Session(engine)
```

Once this all is configured, the user can simply work with mapped objects, instead of data sources themselves. This provides for more readability in the code, where every operation is expressed in the programming language itself, avoiding often complicated operations of SQL-query-building, managing database connection, etc. To perform simple insert and select operations, the user does not need to worry about creating SQL statements, instead, simple manipulation of mapped classes is enough:

```
# insert
user = User(name="sandy", fullname="Sandy Cheeks")
session.add(user)
session.commit()

# select the column value of a specific row in the table
sandy_fullname = session.execute(select(User.fullname) \
                    .where(User.id == 2)).scalar_one()
```

There are several ways to specify the objects representing data sources. Every technology which provides functionality for ORM allows for different approaches and we will talk about them in a greater detail later in this chapter.

4.2 Advantages of ORM

There are several obvious reasons why ORM is a very popular technique among software engineers, for example:

- Simplified development because developers do not have to write code to manipulate data sources directly, but all they need to do is modifying objects in their programming language - the tedious work related to working with databases is handled by the ORM technology used [10].

- Since the ORM-supporting technology handles the majority of data source interaction, the code is simpler, cleaner, more readable, and, as a direct consequence, less error-prone [10].
- Mapping individual pieces of data into domain-model objects brings simplification in developed applications, therefore, faster development and lower development cost [26].
- Increased security - many ORM solutions include data validation and security threat prevention mechanisms which greatly increase application security without any effort [26].

4.3 Disadvantages of ORM

Without a doubt, there are many advantages which make using ORM technique very tempting, however, we shall not forget about the downsides which are often underestimated when a new piece of technological stack is being chosen by developers. Between the biggest ORM disadvantages are:

- Overhead - ORM solutions rely heavily on abstraction and, as it is common for such cases, it introduces some performance penalty. All handling of interactions with data sources requires a lot of overhead which we, fortunately, do not see in the code, but it is present anyway [6].
- Additionally, ORM-supporting technologies do not always use the most optimal SQL queries possible and they cannot be modified. Using embedded SQL in the code, on the other hand, allows for executing any query (even an *invalid* one), providing means for query optimizations in performance-sensitive programs [6].
- Learning how to use a new technique, especially the one that differs to some degree between all technologies supporting it, is somewhat tricky and lengthy. For this reason, many companies decide to opt for the usage of embedded SQL [6].
- Using a technology or any technique which is not understood entirely brings a risk of encountering problems with common pitfalls to the project. In case of ORM, it is, for example, the *lazy loading pitfall* [11].
- Another aspect to consider is complicated situations when several data sources (e.g. database tables) need to be altered. In plain SQL, these situations could be handled by several clauses, however, in ORM, all clauses have to be analogously performed in code, which is often more difficult than writing an SQL query [26].

4.4 Usage

Despite the above-mentioned disadvantages, ORM is still used by developers in various situations. Web applications usually utilize ORM often since it allows for

an easily implemented functionality which persists data in a database and does not require, in most cases, any complex logic.

Thanks to the ORM support of their chosen technology, they can simply focus on implementing business logic without a focus on how data is about to be persisted, resolve how database connection is going to be handled and handle all exceptions that database communication can throw.

In these web applications, usually, only CRUD[1] operations are necessary and, with exception for specific cases, their business logic is rather simple. Using ORM, therefore, simplifies the development and allows for a rapid software development.

When we take into account built-in security mechanisms, ability to generate SQL migration scripts upon model change, or easy configuration, which many ORM-supporting technologies provide, ORM is often the correct choice which makes software development easier and its advantages are greater than disadvantages.

According to the Python Developers Survey 2021 [18], the most popular ORM solutions for Python were SQLAlchemy, Django ORM and SQLObject. In this work, we are going to focus on the most popular solution - SQLAlchemy.

4.5 SQLAlchemy

The SQLAlchemy Toolkit is a set of tools for working with databases and Python. It contains several sets of functionality which can be used separately or together [14]. The whole toolkit is built atop of the DB API v2.0 [15], which specifies a unified way how to access databases in order to ensure a more understandable code, generally portable across databases, and a wider range of supported databases [15].

Note that in this analysis, we are going to focus on SQLAlchemy 2.0-style syntax, which is a newer version of the library. Currently, only SQLAlchemy 2.0 is actively developed, while the 1.0-style is considered as deprecated and only critical bugs are developed for this version.

In general, SQLAlchemy can aggregate its functionality into two major areas of functionality - the SQLAlchemy Core and SQLAlchemy Object Relational Mapper (ORM) [14]. The top-level architecture of the technology can be seen in Figure 4.1.

In Core, there are several standard features that are handy for standard working with databases. The most important is SQL Expression Language, which is a separate toolkit for constructing SQL expressions represented by composable objects, which can be executed against a database within the scope of a specific transaction. It returns a standard result set, common in database libraries of other programming languages as well. Data-manipulating operations, such as inserts, deletes, or updates, are handled by passing SQL expression objects with dictionaries of parameters, which together represent the statement to be executed [14].

On the other hand, the ORM is built atop the Core. The ORM uses Core to represent all database operations that need to be done, providing a domain object model mapping to a database schema. Therefore, the user does not have

[1] CRUD - the four basic operations of software applications: create, read, update, delete

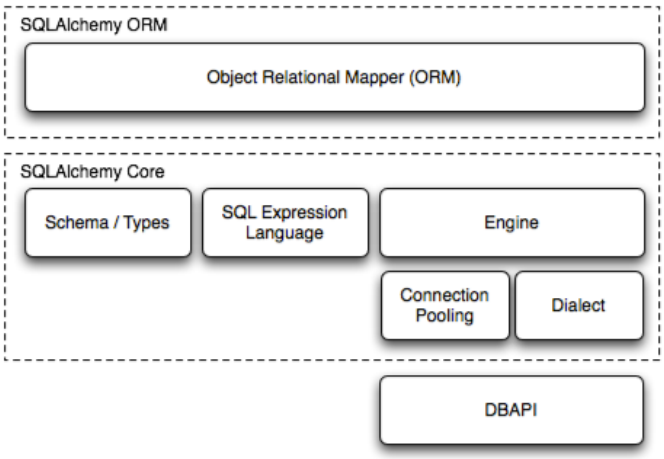

Figure 4.1: A high-level architecture of SQLAlchemy [14].

to use SQL Expressions directly. Instead, the ORM takes care of persisting modifications in business objects (instances of domain object model, representing data from a database) using the unit of work pattern - these changes are automatically translated into the SQL Expression Language and the resulting statements are then executed. Select statements work in a similar manner, used when new business objects are instantiated [14].

When compared, the Core is more command-oriented, relying on immutability and applying a schema-centric view of the database. On the other hand, ORM employs a more object-oriented approach (via domain object model), relying on mutability of business objects, and state-oriented [14].

Because the SQLAlchemy ORM, which is the main topic of this analysis, is based atop the SQLAlchemy Core, it is necessary to explain the basics of the Core module before proceeding with the analysis of SQLAlchemy ORM.

The main object which represents the source of connections to a particular database is called `Engine`. In addition to being a connection pool for database connections, it also serves as their factory. To create a new Engine instance, a user simply needs to provide the URL string which describes how to connect to the database host. It is quite similar to common connections strings [7].

```
url1 = "dialect+driver://username:password@host:port/database"
url2 = "postgresql+psycopg2://scott:tiger@localhost/mydatabase"

first_engine = create_engine(url1)
another_engine = create_engine(url2)
```

When an engine is created, its only purpose is to provide the object serving as the connection, named *Connection*. This new object is used for all database interactions. A simple usage to select all entries from a table and print it can look like this [29]:

```
with engine.connect() as conn:
    result = conn.execute(text("select 'hello world'"))
    print(result.all())
```

Changes are not autocommitted and, therefore, when the connection is released, changes are rolled back. To commit changes, the *commit as you go* approach can be chosen, which requires the explicit commit command to be invoked, or the *begin once* approach can be used as well. In such case, the connection is created by invoking the `begin()` function and all changes are automatically committed when the connection is released, given the code ran without a problem, or it is rolled back in case there was an exception [29].

```
# commit as you go
with engine.connect() as conn:
    conn.execute(
        text("INSERT INTO some_table (x, y) VALUES (:x, :y)"),
        [{"x": 6, "y": 8}, {"x": 9, "y": 10}])
    conn.commit()

# begin once
with engine.begin() as conn:
    conn.execute(
        text("INSERT INTO some_table (x, y) VALUES (:x, :y)"),
        [{"x": 6, "y": 8}, {"x": 9, "y": 10}])
```

Note the usage of the `text()` function which provides a backend-neutral way for parameter binding, per-statement execution options and result-column typing behavior [2].

When executing SQL statements which are expected to return some data, the `Result` object is used, which represents an iterable wrapper around all returned `Row` objects. The Row class implementation is intended to behave very closely to Python's named tuples and, therefore, can be iterated over in several different ways [29], as shown in Figure 4.2.

Database Metadata

If users want to work with a relational database, they usually create and query the basic database object, table, represented by the `Table` class in SQLAlchemy. Before starting the work with *SQLAlchemy Expression Language*, typically, a user may want to have all tables that they are going to be working with represented by Tables [28].

Tables may be declared, when a user defines exactly what the table looks like, or reflected, when the user lets SQLAlchemy to generate a Table based on what is present in the database. Resulting Tables can be worked with together, no matter how they were created [28].

Before a Table can be created, a `MetaData` object, which is a facade around a Python dictionary where Table objects are mapped by their string name, has to be created first. A single MetaData object is commonly used in the entire application (and used as a module-level variable in the `models` or `dbschema` type of package), but there can also be multiple MetaData collections as well. It is typically the most helpful if a series of Table objects related to each other belong to the same MetaData object [28].

```
result = conn.execute(text("select x, y from some_table"))

# tuple assignment
for x, y in result:
    print('x = ' + str(x))

# integer index
for row in result:
    x = row[0]

# attribute name
for row in result:
    y = row.y
    print(f"Row: {row.x} {y}")

# mapping access
for dict_row in result.mappings():
    x = dict_row["x"]
    y = dict_row["y"]
```

Figure 4.2: Different approaches to the iteration of the `Result` object.

A Table object usually consists of `Column` objects, which must be passed to the constructor when creating a new Table declaratively, along with table name and the MetaData object where the new Table object shall belong. These columns are, then, accessible via the `Table.c` field. Various constraints can be added for every column, but that is not important from our point of view.

```
from sqlalchemy import MetaData, Table, Column, Integer, String

metadata_obj = MetaData()

user_table = Table(
    "user_account",
    metadata_obj,
    Column("id", Integer, primary_key=True),
    Column("name", String(30)),
    Column("fullname", String))

# prints "Column('name', String(length=30), table=<user_account>)"
print(user_table.c.name)

# prints "['id', 'name', 'fullname']"
print(user_table.c.keys())
```

Another approach to working with table metadata is to use reflection. This approach means that the Table object is generated according to the current state of a database. While declared tables can be used to emit DDL to the database

(and create database schema according to tables in the MetaData), the table reflection does this in reverse. It can create Table according to database's schema, by specifying the engine to be used, together with the table name [28].

```
some_table = Table("some_table", meta_obj, autoload_with=engine)
>>> some_table
 ...Table('some_table', MetaData(),
    Column('x', INTEGER(), table=<some_table>),
    Column('y', INTEGER(), table=<some_table>),
    schema=None)
```

Note that reflected tables can also override certain columns [20]:

```
my_view = Table(
    "some_view",
    metadata,
    Column("view_id", Integer, primary_key=True),
    Column("related_thing", Integer,
    ForeignKey("othertable.thing_id")),
    autoload_with=engine)
```

Additionally, all database tables can be reflected at once [20]:

```
metadata_obj = MetaData()
metadata_obj.reflect(bind=someengine)
addresses_table = metadata_obj.tables["addresses"]
```

Other approaches to reflection can be also used, such as loading tables from some other schema [20]:

```
metadata_obj = MetaData(schema="project")
metadata_obj.reflect(someengine)
messages_table = metadata_obj.tables["project.messages"]
```

For a fine grained reflection, users may use the *Inspector* functionality. We are not going to get into details of the Inspector's usage as it is not widely used and not really usable in relation to SQLAlchemy ORM.

```
engine = create_engine("...")
insp = inspect(engine)
print(insp.get_table_names())
```

Working with data

Now that we know how metadata work in SQLAlchemy Core, let's have a quick look at how working with data looks like, so that we can better understand the differences and similarities between Core and ORM modules.

As pointed out earlier, the Core module relies to a great extent on its SQL Expression Language. Alongside the traditional raw query execution with parameters (using, for example, the `text()` function, mentioned earlier), equal operations can be executed and expressed by the expression language. This is not the

the most important part of the analysis and, therefore, we will just show simple examples of common CRUD operations.

To insert a single row, the `insert()` function can be used:

```
from sqlalchemy import insert
stmt = insert(user_table).values(name="spongebob",
                fullname="Spongebob Squarepants")
with engine.connect() as conn:
    result = conn.execute(stmt)
    conn.commit()
```

Selection of data is very similar to the insertion - using the function `select()`, a `ScalarSelect` object is created. It supports defining additional clauses and/or parameters by invoking methods over the object, such as `where()` or `column()`. The execution of the select statement then returns a Result object which is iterable, as shown in Figure 4.2.

```
from sqlalchemy import select
stmt = select(user_table).where(user_table.c.name == "spongebob")
with engine.connect() as conn:
    for row in conn.execute(stmt):
        print(row)
```

An interesting piece of functionality is the ability to define a select statement via an ORM entity. The returned data, as opposed to a standard select, contains one ORM entity instance per row, as opposed to a standard Result object, we will discuss more on the topic of ORM later, in Section 4.5.1.

```
# some setup code before

class User(Base):
    __tablename__ = "user_account"

    id: Mapped[int] = mapped_column(primary_key=True)
    name: Mapped[str] = mapped_column(String(30))
    fullname: Mapped[Optional[str]]

row = session.execute(select(User)).first()

# prints:
#(User(id=1, name='spongebob', fullname='Spongebob Squarepants'),)
print(row)
```

There are, of course, many other options that are commonly used with select statements in SQLAlchemy, such as defining the FROM clause, ordering, grouping, joins, or unions. However, this is not our main focus at the moment and we are not going to go into too much detail.

Updates and deletes work in the same manner, providing practically equal functionality that can be achieved using raw querying, but in a programmatic manner.

4.5.1 SQLAlchemy ORM

When compared to the Core functionality of SQLAlchemy, we should not focus on what is present in one and not in the other, but rather what is present in SQLAlchemy ORM in addition to Core, since it is rather a Core's extension.

A fundamental transactional object used in ORM is called `Session`. It is an object that enhances the Connection, which is fundamental for Core and, in fact, the Session refers to a Connection instance internally, which is then used to work with SQL itself [29].

When used in a non-ORM way, the Session simply forwards SQL statements to its internal Connection object and returns what the Connection object does. It also provides Connection's functions, such as `execute()`, `commit()` or `rollback()`. A simple usage in a non-ORM way would look as follows:

```
with Session(engine) as session:
    result = session.execute(text(
        "SELECT x, y FROM some_table WHERE y > :y ORDER BY x, y"),
        {"y": 6})
    for row in result:
        print(f"x: {row.x}  y: {row.y}")
```

Database Metadata

In ORM, key concepts of database metadata mapping around Table, Column and MetaData objects stay more or less the same. One difference is that the MetaData object is commonly associated with an ORM-only construct known as the *Declarative Base* [28]. It can be acquired by sub-classing the SQLAlchemy's `DeclarativeBase` class:

```
from sqlalchemy.orm import DeclarativeBase
class Base(DeclarativeBase):
    pass
```

This `Base` class is referred to as the Declarative Base. Later, when new classes are created by sub-classing the Declarative Base, they are established as ORM mapped classes, usually representing a single Table object.

The Declarative Base refers to a MetaData object which it creates automatically in case it is not provided, as well as the `registry` instance, which is the central mapper configuration unit. They both can be accessed via their property attributes [28]:

```
Base.metadata
>>> Metadata()
Base.registry
>>> <sqlalchemy.orm.decl_api.registry object at 0x...>
```

In SQLAlchemy 1.4, the standard way to create a Declarative Base was to initialize the registry and the MetaData object manually, or to generate both using a standard function `declarative_base()` [27]:

```
from sqlalchemy.orm import registry
mapper_registry = registry()
# MetaData accessed via:
# mapper_registry.metadata
Base = mapper_registry.generate_base()

# alternative to the code above:
Base = declarative_base()
```

The Declarative Base class serves as the base class for all ORM mapped classes that are declared. As of SQLAlchemy 2.0, a common way to declare a class is using the PEP 484 type annotations [16], indicating attributes to be mapped as particular types [28]:

```
class Address(Base):
    __tablename__ = "address"

    id: Mapped[int] = mapped_column(primary_key=True)
    email_address: Mapped[str]
    user_id = mapped_column(ForeignKey("user_account.id"))

    user: Mapped[User] = relationship(back_populates="address")
```

As you can see, not all attributes need to be initialized using an assignment operation - if there are no specific constraints, a simple type definition is sufficient. The usage of type information is not necessary - all this information can also be passed to the `mapped_column()` function and assign the result to a variable. In case of the `id` column, we could equivalently declare the field as `id = mapped_column(Integer, primary_key=True)` which produces the same output as the code on line 6 in the example above.

When the `Address` class is initialized, it collects all available information - table name, columns and foreign keys and uses it to create a new Table instance, equal to the one created in a non-ORM way, using the Table constructor. The new Table instance can be accessed via the `__table__` variable [28].

```
print(Address.__table__)
>>> Table('addresses', MetaData(),
...   Column('id', Integer(), table=<address>, primary_key=True),
...   Column('email_address', String(), table=<address>),...)
```

The declared class has also got an automatically generated constructor used with keyword parameters.

```
my_addess = Address(email_adress="my.email@gmail.com", user_id=5)
```

In SQLAlchemy 1.4, all columns had to be declared using the `mapped_column()` function, however, this approach is not preferred because of the PEP 484 [16] and its better support in IDEs and other tools. It is, however, still supported.

Table declarations and ORM declarative approach can be combined, where the Table is directly assigned to the `__table__` attribute:

```
address_table = Table("address",
    metadata_obj,
    Column("id", Integer, primary_key=True),
    Column("user_id", ForeignKey("account.id"), nullable=False),
    Column("email_address", String, nullable=False))

Base = declarative_base()

class Address(Base):
    __table__ = address_table
    user = relationship("User", back_populates="addresses")
```

This mapping style, named within SQLAlchemy as the *declarative style*, however, is not the only way to define ORM mapping. Another option is to use the *imperative style*.

This style uses imperative table definition, common in SQLAlchemy Core, which then maps imperatively created Table objects to empty classes [12]:

```
mapper_registry = registry()

user_table = Table(
    "user",
    mapper_registry.metadata,
    Column("id", Integer, primary_key=True),
    Column("name", String(50)))

class User:
    pass

mapper_registry.map_imperatively(User, user_table)
```

Working with data

Because the ORM technology is intended to allow for working with data sources in a simpler manner, data manipulation is very straight-forward. Earlier, we have shown how to map database tables and columns to objects - every class represents a table (or a part of a table) and its fields represent columns of the mapped table. Logically, then, one instance of this mapped class is going to represent a single row of the mapped table.

Let there be a table tracking all characters of a cartoon, named `User`, having three columns - `id, name and fullname`. Thanks to the automatically-generated constructor of the mapped class, named User, we can create a new instance as follows [5]:

```
squidward = User(name="squidward", fullname="Squidward Tentacles")
```

You may notice that we did not assign the `id` of the new object. That is because we want to make sure that the primary key of the row, which we would like to insert, is provided using an auto-incrementing primary key feature of the database. Meanwhile, it is represented in the instance as None:

```
print(squidward)
# prints
# User(id=None, name='squidward', fullname='Squidward Tentacles')
```

To start working with the database, a Session object has to be initialized, as mentioned earlier. Once it is ready, the newly created User can be added into the database, using a simple `add()` function.

```
session = Session(engine)
session.add(squidward)
```

It is important to mention that this action itself does not automatically interact with the database. Because the SQLAlchemy ORM uses the *unit of work* pattern, all changes are stored temporarily in the Session object, which keeps a list of changes before it flushes them - writing into the database. Therefore, the data is inserted into the database only when the `session.flush()` command is invoked.

Manually flushing data may sometimes be unnecessary, for example, when the autoflush feature is enabled (enabled by default; all query operations issue a flush command to their related session before proceeding). Changes are also flushed whenever the `Session.commit()` function is invoked.

As it is clear, the insert statement is automatically generated from the modifications tracked by the Session object and the user does not have to write a single line in SQL.

For selecting rows, the process changes a little bit. The Session object provides the `get()` method which takes two compulsory arguments - the mapped class (entity into which the corresponding row shall be mapped) and the primary key specifier, which identifies the row to be loaded. For example, the User with id 5 can be selected very easily:

```
some_squidward = session.get(User, 5)
print(some_squidward)
# prints
# User(id=5, name='some_squidward',fullname='Squidward Tentacles')
```

Now, if a user wants to make some changes to the row, they only need to modify the instance that was returned.

```
some_squidward.name = 'some_squidward2'
```

However, because changes have not been flushed, the changed name is not yet in the database. If the same row over the same session is selected, using the filter for the name that was just changed, we would see that the autoflush functionality would first flush changes into the database and then return the row we changed locally:

```
squidward = session.execute(select(User)
                    .filter_by(name="some_squidward2"))
                    .scalar_one()

# prints True, because both instances contain data
# of the same row in the database
print(squidward is some_squidward2)
```

Therefore, to update data in the database, we selected rows need to be modified in the code.

Lastly, to delete a row, only a simple usage of the `delete()` function is needed:

```
patrick = session.get(User, 3)
session.delete(patrick)

# this select flushes the deletion
session.execute(select(User).where(User.name=="patrick")).first()

# prints False - changes were autoflushed,
# patrick was deleted from the table
print(patrick in session)
```

It is worth mentioning that even though the data was flushed into the database, it was not yet committed. To commit the transaction which was started automatically by performing changes using the ORM approach, `Session.commit()` function needs to be invoked. In such case, all objects that are being used in the program are still valid, but any subsequent change would initialize a new transaction. Alternately, `Session.rollback()` can be invoked to revert all changes.

In some cases, the ORM approach to working with data is not suitable, for example, if a user wants to insert a lot of data. Initializing a new mapped object only to immediately insert it and never use it again can have a significant impact on the performance of software and, therefore, it is preferred to use a non-ORM approach via the `Session.execute()` functionality, as was already shown earlier and providing arguments in the second function's parameter [13]:

```
from sqlalchemy import insert
session.execute(
    insert(User),
    [
        {"name": "spongebob", "fullname":"Spongebob Squarepants"},
        {"name": "sandy", "fullname": "Sandy Cheeks"},
        {"name": "patrick", "fullname": "Patrick Star"},
        {"name": "squidward", "fullname": "Squidward Tentacles"},
        {"name": "ehkrabs", "fullname": "Eugene H. Krabs"},
    ],
)
```

This single command transforms to and executes the following SQL query:

```
INSERT INTO user_account (name, fullname) VALUES (?, ?)
[...] [('spongebob', 'Spongebob Squarepants'),
       ('sandy', 'Sandy Cheeks'),
       ('patrick', 'Patrick Star'),
       ('squidward', 'Squidward Tentacles'),
       ('ehkrabs', 'Eugene H. Krabs')]
```

Finally, before the work with the Session is terminated, it needs to be closed, by invoking the `Session.close()` function, which releases all connection resources to the connection pool and expunges all objects from the Session [5].

Relationships

Last, but not least, it is important to understand how table relations are defined in SQLAlchemy ORM. This is particularly useful when two or more tables are associated, for example, when there is a one-to-one relation, where a row in a table references another row in a different table via a foreign key. In such cases, SQLAlchemy resolves which tables are related and, therefore, which table can be accessed from some other table [1].

The standard way to declare a relationship between two tables is to use the `relationship()` function. This function can be used in both imperative and declarative approaches, both with annotations (using PEP 484 [16]) and without them. By default, this function returns a `List` of related class' instances, which can be configured to return a single value, or a different collection of instances (set, dict, etc.) [1].

Using the most modern approach with annotations, declaring a relationship between two tables, and, therefore, access record(s) of one table from another table, would look like this [1]:

```
class Parent(Base):
    __tablename__ = "parent_table"

    id: Mapped[int] = mapped_column(primary_key=True)
    children: Mapped[List["Child"]] = \
                relationship(back_populates="parent")


class Child(Base):
    __tablename__ = "child_table"

    id: Mapped[int] = mapped_column(primary_key=True)
    parent_id: Mapped[int] = \
            mapped_column(ForeignKey("parent_table.id"))
    parent: Mapped["Parent"] = \
            relationship(back_populates="children")
```

As you can see, the `child_table` table contains a foreign key to `parent_table` table and to be able to work with the related table in an ORM style, a way to access the class instance defined by the foreign key is needed. It is possible via the already-mentioned `relationship()` function which synchronizes the `Parent` class with the `Child` class. Using the `back_populates` parameter, it is possible to specify with which other table's variable shall this instance be synchronized (as seen in the example, the Parent instance is synchronized with the Child.parent attribute, and Child instance is synchronized with the Parent.children attribute.

Using annotations, it is possible to define whether a single or more instances are accessible (one to many or one to one relationship).

In case of `Mapped[List["Child"]]`, a Parent may reference many Child instances, but `Mapped["Parent"]` signs that every Child record may only have a single parent. To change the collection in a 'many' relationship, the annotation can be simply changed to `Mapped[Set["Child"]]` or a similar value. For *nullable* single-relationships, `Optional` shall be used: `Mapped[Optional["Child"]]`.

Using `relationship()` in both related classes allows for a simply-configurable bidirectional access.

In case of the non-annotated declarative style, the same result can be achieved using similar code [1]:

```
class Parent(Base):
    __tablename__ = "parent_table"

    id = mapped_column(Integer, primary_key=True)
    children = relationship("Child", back_populates="parent")


class Child(Base):
    __tablename__ = "child_table"

    id = mapped_column(Integer, primary_key=True)
    parent_id = mapped_column(ForeignKey("parent_table.id"))
    parent = relationship("Parent", back_populates="children")
```

As seen in the code, the approach is very similar. The main difference is that instead of defining the related class in an annotation, the first-position argument of `relationship()` has to be used - this parameter can be either the name of a mapped class or direct reference of the class, such as `relationship(Parent)`. Another difference is that changing the returned collection of many-relationship is done via the function's `collection_class` parameter.

Lastly, the imperative approach is very similar [1]:

```
user_table = Table("user",
    mapper_registry.metadata,
    Column("id", Integer, primary_key=True),
    Column("name", String(50)),
    Column("fullname", String(50)),
    Column("nickname", String(12)))

class User:
    pass

mapper_registry.map_imperatively(User, user_table,
    properties={"addresses": relationship(Address, backref="user",
                order_by=address.c.id)})
```

As seen from the example, the usage of the `relationship()` makes the whole concept reusable in different approaches and the only difference is how the information is defined - whether in the class configuration and its annotations, or using a function explicitly (in *declarative style*, the most work is done in the same way, but functions are invoked implicitly). This way, different relations can be modeled with only slight changes - one to one, one to many, many to one, or many to many.

5. Analysis

Based on the description of technologies that we are going to work with (Python scanner, PySpark, ORM), we are going to focus on the problem of data lineage analysis for these technologies in a more specific manner in this chapter. We will outline the important parts that need to be implemented in order to generate data lineage graphs and we will also discuss the most important requirements for the solution.

Before we start analyses of individual features, we first need to define the scope of this work. To support the PySpark library within the Python scanner, we need to create a new scanner plugin for it. The PySpark plugin, which is a set of *propagation modes* defining how to handle specific PySpark function invocations, needs to support some elementary functionality:

- Initializing a new SparkSession
- Defining a DataFrame schema
- Creating a new DataFrame instance by loading data from a CSV or JSON file, or a JDBC connection, including the generic `load()` function
- Saving a DataFrame instance into a CSV or JSON file, or a JDBC connection, including the generic `save()` function
- Using SQL queries to work with data (`SparkSession#sql()`)
- Data-modifying operations over DataFrames:
  - `crossJoin()`
  - `drop()`
  - `join()`
  - `select()`
  - `union()`
  - `unionAll()`
  - `withColumn()`
  - `withColumnRenamed()`

We will, therefore, focus on the *Spark SQL* functionality, not taking into consideration *Spark Streaming* and *Spark MLlib*.

Since the majority of functionality is performed over DataFrame columns, rather than DataFrames themselves, it is also necessary to support column recognition in this plugin. The Python scanner is implemented to support resource[1] recognition only, so it is necessary to modify how the scanner works in general.

As for the ORM support, we only need to design data structures which are going to represent the ORM defined in the analyzed application. Because the implementation of the ORM support would span long past the scope of a master's thesis, we did not implement the solution within this work.

[1] In the context of the Python scanner, resources are files, database tables, or the console.

Therefore, in the beginning of this chapter, we are going to analyze how the scanner can work on the column level and which constraints there are. Then, we will analyze PySpark's behavior and define what functionality we are going to need from data structures representing the library's objects and briefly describe the behavior of functions which we intend to support in the plugin. In the end, we will discuss what is required from the ORM support design.

5.1 Column Handling

As mentioned above, the scanner does not support column recognition except for a single case - when a query is executed and the `__getitem__()` function is invoked over the result: `result['my_column']`. In such case, *deduction* is used, creating a named column (column named *my_column* in the previous example).

We need a solution that would let us work with columns effortlessly, ideally, without having to know whether a column originates in a file, a database, or if it was created in the program ("hard-coded" in the source code; not being loaded from anywhere). To meet MANTA's standards, columns may be either named, indexed or unknown. Additionally, if a column has both name and an index, for example, when it is know that a column X is n-th in a data structure (the name is X and the index is n), then index is preferred.

Deduction is desired, too. The same way it is implemented for database results already, similar behavior should be implemented for files. As for the third resource supported by the Python scanner, the console, column deduction is not required because it makes no sense - a data line which is read or printed has never got any name or an index.

Another useful feature is tracking origin. Even though the Python scanner currently does not use or visualize which transformations are performed inside the program, only input/output edges, this information may be useful in the future. If we consider data lineage as illustrated in Figure 5.1, according to current scanner implementation, only green columns would be visualized and inner PySpark columns would be removed by contraction of edges. However, it is desired that the new solution keeps track of the whole lineage, allowing the scanner to change the output graph by a simple graph transformation modification, not needing to change how the scanner's data lineage analysis works.

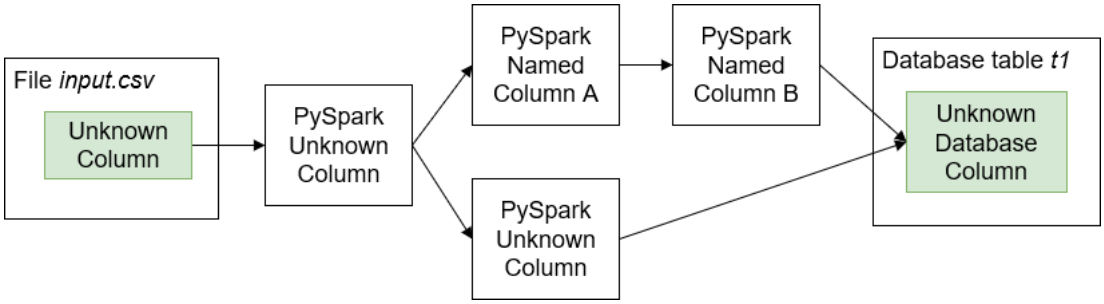

Figure 5.1: An example of a complete data lineage of a Python program. Currently, only green columns are visualized in MANTA.

For cases when a column is recognized, it would be very convenient to be able to add new resource columns without knowing the concrete type of resource.

This can be useful when a function accepts both JDBC connection string or a file path as the resource identifier. When identifiers are recognized and resolved, it is preferred to work with resulting *resources*, not with a file and a JDBC connection separately. For this, a simple interface is a clear solution, but we will talk more about this in the next chapter.

To demonstrate a functioning solution for file columns, we decided to add support for reading and writing columns to/from a CSV file, which will allow us to deduce both indexed and named columns for files.

5.1.1 CSV I/O reading operations in Python

In this section, we will describe the basic reading functionality of CSV I/O operations in Python's built-in library, so that the reader can understand implementation details that we are going to describe in following chapters. All functionality is aggregated in the `csv` module, which is imported by a simple `import csv` command.

For reading, the module provides two reader objects - a simple one, which we are going to call a *common reader*, allowing us to specify columns by index, and a *dictionary reader*, which allows us to navigate across columns by their given name.

Common reader

A common reader instance can be obtained by invoking the following function:

```
csv.reader(iterable, dialect='excel', *args, **kwargs)
```

It returns a reader object over the iterable provided (can be, for example, an opened file, a string, a collection). This iterable is, then, returned upon every invocation of `__next__()` function (one row/iterable item). Before returning a row or other item, the reader first splits the object by a comma separator (or any separator assigned in the reader function invocation) and returns a list of strings resulting from the split operation.

For a string row `'a,b,c'`, it returns a list with 3 string objects `['a', 'b', 'c']`.

The work with these objects is similar to any list. If a user wants to print the second row item, they can simply use the following code:

```
with open('file.csv') as csv_file:
    csv_reader = csv.reader(csv_file, delimiter=',')
    for row in csv_reader:
        print(row[1])
```

A more general use case can be:

```
import csv
with open('input.csv', newline='') as csvfile:
    spamreader = csv.reader(csvfile, delimiter=',', quotechar='|')
    for row in spamreader:
        # returns a list by comma-splitting a row
```

```
        print(row[0])
        print(',',.join(row))
```

Even though the function definition (signature with an empty body) is present in the Python's built-ins file `_csv.py`, the implementation is hidden and not many details are provided in the documentation.

We are mostly interested in the way it works if the *iterable* parameter is a file. We want to be able to see in the data lineage graph that an unknown column was loaded from a path-identified file and printed to a console. If we take, for example, the program below, the expected data lineage visualization is depicted in Figure 5.2.

```
with open("sample.csv") as csv_file:
    reader = csv.reader(csv_file, delimiter=",")
    for row in reader:
        print(row)
```

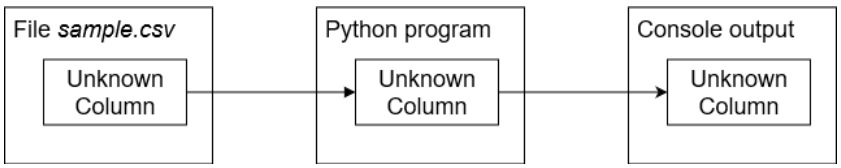

Figure 5.2: Expected output in the data lineage graph of a program which reads CSV data from a file and prints them to the console.

Dictionary reader

An instance of the dictionary-like reader of the built-in library can be obtained, as opposed to the common reader, by simply using its initializing function:

```
csv.DictReader.__init__(self, f, fieldnames=None, restkey=None,
               restval=None, dialect="excel", *args, **kwds)
```

It is an alternative to the index-based reader and, as a matter of fact, the DictReader class is a wrapper around the common reader. The DictReader creates and keeps an instance of the common reader inside and uses that to read columns, transforming columns into dictionary using the *fieldnames* information. If a column is accessed by its name, the DictReader uses its internal mapping to map the name to the column index and returns the common reader's output for the mapped index. If there are no explicit fieldnames provided, the first row is used as the header with names.

An example usage is similar to that of the common reader:

```
import csv
with open('names.csv', newline='') as csvfile:
    reader = csv.DictReader(csvfile)
    for row in reader:
        print(row['first_name'], row['last_name'])
```

Therefore, working with common and Dictionary readers is similar, the only difference is that common one uses integers indexes, while Dictionary reader uses strings as indexes.

In case when no columns are defined in a `__getitem__()` manner, then the expected data lineage graph looks exactly as the expected output in Figure 5.2.

5.1.2 CSV I/O writing operations in Python

Similarly to reading options, Python supports both indexed and named column-writing options.

Common writer

An instance of the common writer is retrieved similarly to the common reader:

```
csv.writer(fileobj, dialect='excel', *args, **kwargs)
```

The common writer uses the *fileobj* parameter as the output resource of its writing operations - this parameter does not necessarily have to be a file, but it can be any object that supports the *File API*.

The function definition is present in the built-in library's file `_csv.py`, implementation is hidden and not many details are provided in the documentation about how the writer works internally.

When writing objects, there are two options:

- Using the function `writerow(self, rowdict)` which writes a single row dictionary - in case of the common writer, it uses a list, a set, or any other iterable object, and writes the content into the file using the rowdict parameter iteration, adding the separator in between items (default separator is comma).
- Using the function `writerows(self, rowdicts)`, similar to the function `writerow(...)`, its parameter *rowdicts* is just an iterable of rowdict, the implementation flattens rowdicts and writes them into a file sequentially.

Example usage of the `writerow(...)` function for writing a single line into the file:

```
import csv
with open('eggs.csv', 'w', newline='') as csvfile:
    spamwriter = csv.writer(csvfile, delimiter=' ',
    quotechar='|', quoting=csv.QUOTE_MINIMAL)

    spamwriter.writerow(['Spam'] * 5 + ['Baked Beans'])
    spamwriter.writerow(['Spam', 'Lovely Spam', 'Wonderful Spam'])
```

And the `writerows(...)` function, in this case, the data contains two lists, therefore, two rows are written, each with four columns:

```
data = [
    [input(), 28748, 'AL', 'ALB'],
    ['Algeria', 2381741, input(), 'DZA']
```

```
]

with open('countries.csv', 'w', encoding='UTF8', newline='') as f:
    writer = csv.writer(f)
    writer.writerows(data)
```

For this example, the expected data lineage output can be seen in Figure 5.3. Note that only two columns use the console input (function `input()`) and, therefore, only those rows have got the console-input data source.

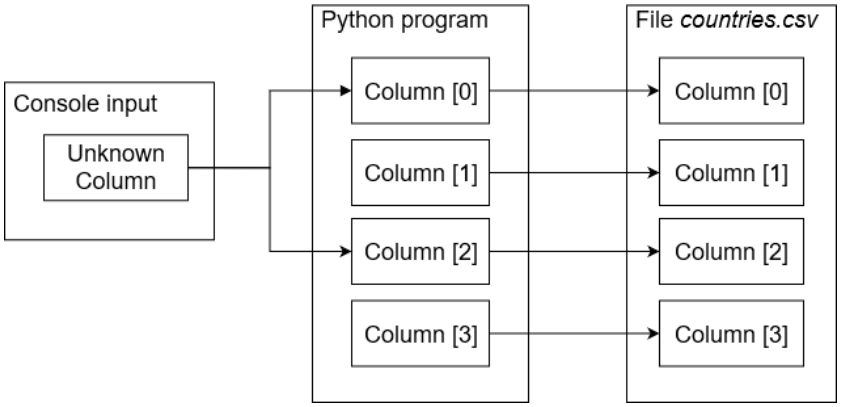

Figure 5.3: Expected output in the data lineage graph of a program which reads console for 2 column input data and print four-column rows to a file.

Dictionary writer

A very similar class to common writer, again, DictWriter uses the common writer for actual operation. This class only translates named indexes to integer-based indexes. An instance can be retrieved by invoking the class constructor:

```
class csv.DictWriter.__init__(self, f, fieldnames, restval="",
            extrasaction="raise", dialect="excel", *args, **kwds)
```

The `DictWriter` provides two writing functions as does the common writer - `writerow(...)` and `writerows(...)`. Instead of lists, dictionaries are provided as items of the provided iterable, they are then mapped back to correct positions during writing operations. Its usage is very simple:

```
import csv
with open('names.csv', 'w', newline='') as csvfile:
    fieldnames = ['first_name', 'last_name']

    # fieldnames specifies what the file header should be,
    # compulsory parameter
    writer = csv.DictWriter(csvfile, fieldnames=fieldnames)

    # arbitrary order of columns - writer resolves keys
    # to correct order using the fieldnames parameter of __init__
    writer.writerow({'first_name': input(), 'last_name': input()})
```

For this example, the expected data lineage graph is very similar to the one of the common writer, with column names changes from indexes to names, as seen in Figure 5.4.

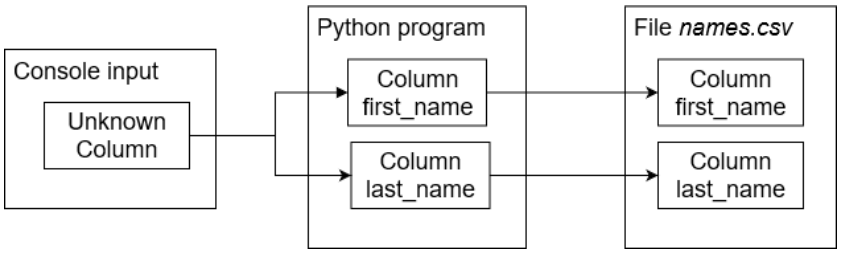

Figure 5.4: Expected output in the data lineage graph of a program which reads console for 2 columns and prints them to a file, specifying their column names.

Having defined requirements for the key feature of extending the Python scanner, which is going to let us analyze PySpark data lineage on the column level, and choosing CSV functionality of Python to verify a working solution verifiable in the output graph, we can move forward to the analysis of PySpark functions which were chosen to be supported.

5.2 PySpark SQL

In order to accurately track all data flows and external data connections, the PySpark plugin would have to be able to find a way to persist all information about available views, data sources and their names or aliases. From the above described PySpark structures, behavior, and relations, it is clear, that supporting all this functionality would require a lot of additional and non-trivial implementation, which would also require many changes in other technologies, for example the Java scanner, which already supports Spark, or the Apache Hive scanner. For this reason, we are not going to focus on supporting tracking sessions and session objects in this work.

5.2.1 DataFrames, transformations and actions

Following up on the Section 3.4.1, we are going to have to design a convenient data structure to represent DataFrames. What we need from DataFrames is to be able to track columns which are present in the 'current' DataFrame. That is, if a DataFrame 1 is created by loading from a file, then DataFrame 2 is created by adding a new column X to DataFrame 2, DataFrame 3 originates in DataFrame 2 and renames column Y to Z, and DatFrame 3 is written into another file, we want to be able to have all this information present in data lineage.

This example with DataFrames 1-3 can be translated into following source code:

```
spark = SparkSession.builder.getOrCreate()
df1 = spark.read.csv("./input.csv")
# new column 'X' has always got value '1'
df2 = df1.withColumn('X', lit(1))
```

```
df3 = df2.withColumnRenamed('Y', 'Z')
df3.write.csv("./output.csv")
```

The expectation from the processing of DataFrame operation is that we can retrieve as much information from the source code as possible about the structure of data. Therefore, we want the PySpark plugin to be able to identify that:

- DataFrame 1
  - originates in file *input.csv*
  - contains only a single unknown column
- DataFrame 2
  - originates from DataFrame 1
  - contains an unknown column and a column 'X'
- DataFrame 3
  - originates from DataFrame 2
  - contains an unknown column, column 'X', and column 'Z'
  - these three columns were written to file *output.csv*
- Since DataFrame 2 contains column 'X' and DataFrame 3 rename 'Y' to 'Z', that means that DataFrame 1 contained column 'Y', therefore, *input.csv* contained an unknown column and column 'Y'

We need the DataFrame to be able to track all these changes and, thanks to the column recognition design, this capability shall be available for users of the scanner. As we explained at the end of Chapter 2, only input/output points of programs are visualized - internal data flow edges are contracted and, therefore, for our example with DataFrames 1-3, the data lineage graph shall resemble the graph in Figure 5.5. Therefore, only known columns of the *input.csv* and *output.csv* are to be visualized, with edges correctly assigned between columns of these two files.

Similar approach stands for other transformations which selected to be supported in the PySpark plugin at the beginning of this chapter. In terms of actions, we only outlined I/O operations to be supported, which we discuss in the next section, with focus on database data sources.

However, this is not the only problem related to DataFrames. It is also necessary to keep in mind that DataFrame schema is going to have to be tracked, for example, when a DataFrame is loaded from a data source with pre-defined schema. Luckily, this is not much of a problem and we can approach schema as a DataFrame with no data source. Therefore, a `StructField` object is equal to a DataFrame's column and the `StructType` object is DataFrame. All we need to do is if a schema is provided, to correctly match data source to schema, by matching columns to their sources.

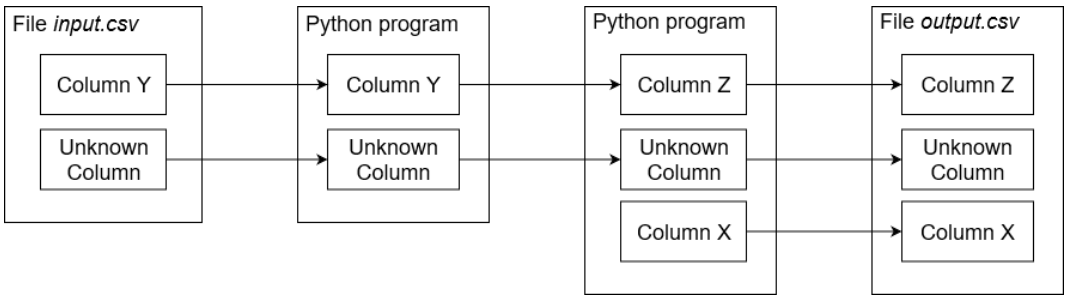

Figure 5.5: Expected output in the data lineage graph for the example with DataFrames 1-3.

5.2.2 I/O operations

Even though our implementation is not going to be able to track all catalog and runtime state information, it does not automatically mean that a simplified solution could not exist for less complicated use cases. In this section, we are going to follow-up on the PySpark I/O operations from Section 3.4.3 and focus on some database data source specifics, as it is a more complex topic than file system resources, and which we are going to need to resolve in our design. We will discuss how DataFrames are initialized from JDBC connections, what are some of its implementation specifics, and what to look out for. We are not going to further analyze file data sources in-depth as they are not as complex as database data sources and there is, therefore, no need to dig deeper into this topic.

As we already described in Section 3.4.3, there are three basic ways how to load DataFrames from a JDBC connection, which do the same thing on the background, and those are DataFrameReader's functions `load()`, `jdbc()` and `table()`. Analogically, for writing, the three options which are the same on the background are DataFrameWriter's functions `save()`, `saveAsTable()` and `jdbc()`.

While read/write operations with file resources only require the path to the file provided, it is different for database sources. To define which JDBC connection, schema, and table shall be used, PySpark requires two properties. First of all, a JDBC connection string must be provided to the database source which should be used. This specifies which DBMS is to be used and when a user wants to specify which table is to be used, there are two mutually exclusive options:

- Provide the name of the table to be used. If schema is used, this option can be in format `<schema>.<table>`, otherwise a simple table name can be used. PySpark, then, returns a DataFrame representing the loaded table. The key for this option is `dbTable`.
- User provides a query to be used in the FROM clause of the SQL query. PySpark then adds parentheses around the query and uses in the template `SELECT <columns> FROM <query> spark_gen_alias`. The key of this option is `query`.

The resulting query is then executed, retrieved data are stored in a new DataFrame instance and the user can continue working with it.

In case of writing a DataFrame, PySpark, again, creates a query that would write the data into the desired database, takes into account the *mode* option

configured (e.g. *append* or *overwrite*) and executes it. In this case, the option `query` makes no sense and is not allowed.

However, for interacting with databases and executing SQL queries, there is one more option for inter `SparkSession.sql(query)`. This function executes the query, provided as a string, over the configured data source and returns a DataFrame containing the result data. However, this function does not limit users to only use it for SELECT statements. In case of a query which does not return any data (INSERT, CREATE TABLE, etc.), an empty DataFrame object is returned, which can be ignored. Spark contains an internal parser for these queries which it processes and executes. By standard, Spark supports Apache Hive, DB2, Derby, H2, MS SQL, MySQL, Oracle, PostgreSQL, and Teradata dialects.

It is also worth noting that in case of the `sql()` function, not only database resources need to be used in the FROM clause. Any resource that can be turned into a DataFrame is accepted, for example, a path to a PARQUET file is accepted. Then, before execution, Spark loads the file into the memory and executes the query over it. Unfortunately, not many details are provided in the documentation about all supported options, and therefore, it is not possible to clearly specify where are the limits of capabilities of the `SparkSession.sql()` function.

This all means that we need to be able to track all configuration used for reading and writing resources in relation to DataFrames. The option-setting functions can affect how PySpark behaves and, therefore, we must be able to analyze this configuration, together with format-specific reading and writing functions' configurations in order to create a reliable data lineage graph. Therefore, if a user sets the format, path, or database table option, we must keep this information for later usage, when a reading/writing operation, such as `load()` or `save()` is executed.

Additionally, specifically for JDBC connections, we are going to need to also count with an option that a JDBC connection information is going to be unknown. While for file system resources it is not a problem, since it is handled in general by MANTA's component named Node Creator, there is no such universal solution for database data sources. If, for example, the scanner encounters a `SparkSession.sql()` function with no database connection details, it cannot just skip this situation. Instead, a solution is necessary that would allow the user to provide additional 'fallback' information.

5.3 Flow variables

We have already mentioned a limitation of the Python scanner and its approach to processing library function invocations in Section 2.2.3. When we are analyzing them, our main focus is to process (or, as it is common to call, propagate) data flows form the source of the function to its target (output) in a specific manner which resembles the actual function behavior translated to data structures used in the scanner.

If there is a file path represented as a string on the input to the propagation mode processing reading from a file (e.g. `readlines()` invoked on a file instance), a new `FileReadFlow` instance is created, which represents the output of this operation. The flow knows the path it *read* the data from and even though the

exact structure of read data is unknown, it is possible to continue the computation with this result and produce reasonably precise results.

However, there are some cases when this is not enough and more details about the flow objects which represent an entity in the program runtime are needed. Since we already talked about PySpark's DataFrame readers and writers, we can use a simple PySpark code to illustrate the issue and show what happens in the current implementation.

A simple use case

Let us have a simplified program which uses the default SparkSession to create a DataFrame by reading a CSV file and then writes it into another file:

```
spark = SparkSession.builder.getOrCreate()
df = spark.read.csv("./input.csv")
df.write.csv('my_file.csv')
```

In the code snippet above, every line contains the problematic attribute access of a library-object instance. On line 1, it is `SparkSession.builder`, on line 2, it is `spark.read`, and on the last line, it is `df.write`. What happens is that when the Python scanner tries to resolve what data flows are assigned to expression `spark.read`, where `spark` is the result of the invocation of `getOrCreate()`, it finds out that there are no flows.

The reason is that the assignment on the line 1 only deals with an assignment to the `spark` expression and there is nothing about `spark.read`, since the assignment to the `read` attribute happens in the library source code, which is not analyzed command-after-command. As a matter of fact, in this case, there is not even an assignment to the `read` attribute, because it is handled by the `@property` decorator.

Nevertheless, the scanner is unable to find any flows belonging to the expression `spark.read` and, therefore, returns an empty collection of flows related to it. In case of lines 1 and 2, this does not have to be an issue, because no input flows are needed to create a session or a DataFrame instance (considering the scope limitation we set earlier about not tracking data related to session instances). For creating a DataFrame instance, only the `"./input.csv"` parameter is important because it provides the path to the file whence the data is loaded.

It is, however, a problem on the line 3, because the scanner must know which DataFrame is being written into the file. The DataFrameWriter keeps a reference to the DataFrame instance it writes. However, if `df.write` returns an empty collection of data flows, the scanner has no idea which DataFrame should be processed during the propagation of function `DataFrameWriter#csv`. As a result, no DataFrames are written and an important part of data lineage is lost.

A general use case

This problem can be observed, in general, in a single situation: during invocation of functions of library object's attributes when the invocation target (usually named the `self` parameter) plays its role. Whenever there is a command, such as `my_var = obj.field.foo()`, and `foo()` utilizes the `self` parameter, it is necessary to resolve the field correctly - `obj.field` may already be known, because

it was assigned explicitly, but it may also not have been used before, and the `obj.field` is not in the collection of known expressions yet.

To resolve this problem, a new feature of the scanner must be designed and implemented that would allow developers to avoid this unwanted behavior and help the scanner to propagate flows correctly.

5.4 Object-relational Mapping

In Chapter 4 we explained details about ORM and SQLAlchemy ORM and now we can start with the actual analysis - what is the information we need to track, propagate and use for data lineage?

First and foremost, the scanner must be able to keep information about which table and columns are mapped by a specific class. It certainly does need to know that a class `A` has got columns `x, y, z` and represents a table named `my_table`. This information needs to be available for the mapped class, since the class reference is passed for selects, specifying to which class should selected rows be mapped.

For simple column initialization, the procedure would be easy, as `Column.__init__()` and `mapped_column()` functions are used in imperative and declarative approaches, respectively. However, SQLAlchemy allows for the implicit field declaration, using only the type hint: `column : Mapped[type]`. This would pose a major challenge to handle as there is practically no explicit invocation which means that no propagation process is launched.

Per column, only the name of the table's column that is mapped is needed to be known (the table shall be its parent and the column itself probably won't need to know its parent). Information whether a column is a primary key or what type the data is in it, is irrelevant for us.

However, at the moment, the Python scanner does not fully support *variable descriptors*, objects representing variables declared in classes, module and functions. This is a major limiting factor in gathering column information for these classes. When a new class definition is encountered in the code, a `ClassDefinitionFlow` object is created to keep information about a class being present in the context of the module it is defined it. This class keeps reference to the associated `ClassDefinition`[2] object, which could theoretically provide information about which variables are defined in the scope of the class, though it should, preferably, be done differently. The scanner should be able to fully support variable descriptors, which could, then, be associated to a class, similarly to how functions can currently be associated to their parent class.

Subsequently, the `ClassDefinitionFlow` object would have to contain a reference to its related `ClassDescriptor` instance, rather than the raw Program Model's `ClassDefinition` object. This way, the class descriptor would provide much more information in case it was passed as an argument, for example in the following code:

[2] A `ClassDefinition` is an object originating in the Program Model, which originates in a class definition directly in the parsed source code. It represents the code structure of the class and its functions, variables, and nested classes, as present in the source code.

```
# User is a class used for mapping
select(User).where(User.name == "patrick")
```

Another aspect to consider is mapping relationships between tables. As summarized at the end of Section 4.5.1, SQLAlchemy allows for defining relationships, based on foreign keys. This relationship can be defined in several different ways, but mostly, the related column is specified via a string value which is then processed by the registry to correctly match columns.

This means, that we would have to collect all mapping classes together in order to properly resolve columns and objects - mapping to class A might use columns from class B if A has a foreign key reference to the table mapped by class B. The whole registry workflow needs to be created in order to provide the global information. This problem is similar to the problem of Spark context in the PySpark plugin - there, the `SparkContext` and the `SparkSession` instances need to be available as well.

The relationship information is very important from the data lineage point of view because this mechanism allows users to access different table-mapping classes from one another (e.g., we can access the Address table from the User table). For such cases, when the scanner encounters it, it is essential to be able to determine which column from which table is used and to accurately track this information. A good example is when the related class is modified via the instance of some other class:

```
class Parent(Base):
    __tablename__ = "parent_table"

    id: Mapped[int] = mapped_column(primary_key=True)
    name: Mapped[str]
    children: Mapped[List["Child"]] = \
                    relationship(back_populates="parent")


class Child(Base):
    __tablename__ = "child_table"

    id: Mapped[int] = mapped_column(primary_key=True)
    parent_id: Mapped[int] = \
                mapped_column(ForeignKey("parent_table.id"))
    parent: Mapped["Parent"] = \
                relationship(back_populates="children")


# ... session preparation ...

child = session.execute(select(Child).where(Child.id == 2))
                .scalar_one()
# this code works with Parent, but the Child is queried
child.parent.name = 'changed_name'
```

An another problem we need to look into is inheritance. Declaratively mapped classes are defined via inheritance of the declarative base, which takes care of registering these classes to the registry. However, the Python scanner currently only supports the inheritance to the level where the invocation lookup is done over functions defined in a class and its predecessors.

All mapped classes use the declarative base's constructor, so this may be useful for our case - since the declarative base is essentially a class generated by SQLAlchemy dynamically (during the registry initialization), which transforms the keyword arguments passed into it to constructed class' attributes. However, this poses another problem for us - the way the Python scanner works, it needs a static-code-present function to define its plugin behavior. If a way is found to map the dynamically created declarative base's `__init__()` function to the inheriting mapped classes, the resolution of this problem would possible.

Lastly, it is also need from the scanner to be able to work with type annotations, since it is the latest approach to defining the ORM model. Without this ability, annotation-based mapped classes would be virtually unusable as the scanner would not be able to recognize fields as database-column-mapped entities.

Due to a large amount of unsupported features necessary for the implementation, we decided to only design the solution of the ORM support feature without the implementation, which would greatly extend this work past its original scope.

The design, however, contains a precise solution to all highlighted problems and ORM features that need to be implemented in the Python scanner, should the scanner be able to analyze data lineage in the source code which uses the ORM technology.

6. Design

In this chapter, we are going to introduce, discuss, and reason about the design of solutions to individual problems outlined and analyzed in previous chapters.

6.1 Column Handling

Before we can start with the implementation of the PySpark plugin or the support for column recognition in CSV file operations, we first need to design how is the whole column handling going to work. In Section 5.1, we concluded that the column-handling solution needs to have following properties and capabilities:

- easy-to-use interface,
- tracking of column origin,
- column deduction,
- working with columns without knowing concrete resource type.

To ensure that all requirements are fulfilled, we split the design of this feature into several parts, each focusing on a certain aspect of data lineage analysis in relation to data columns.

In Sections 6.1.1 and 6.1.2, we talk about the overall design in relation to the data life cycle and decisions which ensure that the design is easy-to-use and easy-to-understand. In section 6.1.3, we discuss the problem of tracking the column origin and Section 6.1.4 explains how deduction is to be performed in the proposed design.

We disclose parts of the design that allow us to ignore the column resource type when working with columns in Section 6.1.5. Because it is necessary to transform the output of the data lineage analysis into a format understandable by the *Dataflow Generator*, adjustments in the transformer had to be designed as well, about which we talk in Section 6.1.6. Lastly, in Section 6.1.7, we summarize the whole design of this feature.

6.1.1 Data life cycle

When we look at the way that column flows are going to work, it is always the same scenario. As we can see in Figure 6.1, the life cycle of every piece of data in Python programs can be summarized in three steps:

1. **Load data** - get the data to work with. This step can be omitted from our point of view when the data is hard-coded into the source code. In such case, no data flows are present, because there is no interaction with external data sources.
2. **Work with data** - perform transformations, move the data across functions, etc.

3. **Write the data** - transformed and/or modified data is written to a resource, for example, to a database. This step is important, because without a write operation, it makes little sense to track inputs.

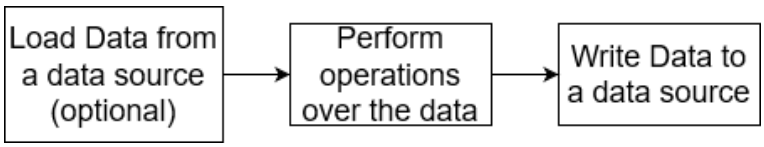

Figure 6.1: A very simplified life cycle of data in Python programs.

When we expand this idea a little bit, we can say that at the beginning of every data life cycle is a data resource which exists independently on the Python program being analyzed. When we read data from it, we load *data columns* from it. Every input can be categorized into three types of columns:

- **Indexed columns** - data that can be specified by providing its numeric position are called indexed columns in the context of MANTA. If a user selects, for example, the third column from the input of a CSV file or they write the second column of a PySpark DataFrame, we talk about indexed columns. Indexed columns are superior to named columns in data lineage graphs, if a column has both a known name and a known index.
- **Named columns** - data that can be specified via a string name. For example, selecting column `user_id` uniquely names a column *user_id*.
- **Unknown columns** - data which has no identifier. This is for cases when we are unable to identify data. This can be, for example, a data line from console input, or unknown columns loaded from a file or a database table. During the analysis, we may get find out more information about columns of a file or a database table, but at the moment of reading data, we usually have no information about the data and its identification.

This data can be, then, represented internally in the program in any way. At the end of the life cycle, again, we have some data, which we can categorize into columns, and write them into a data resource (see Figure 6.2).

Figure 6.2: Expanded life cycle of data in Python programs.

From the point of data lineage analysis, it makes the most sense to try and represent data flows in a way which resembles the actual state of the program for the sake of simplicity and avoiding problem in complex contexts. Therefore, we are going to try to replicate this life cycle with our design.

This means that in our data flows, we are going to have to distinguish between three different entities, representing three stages of data, as we have shown in Figure 6.1. Even though the beginning and the end of the life cycle relate to

data resources, their roles are completely different - while the data resource at the beginning of the life cycle provides base for a data flow which is to be propagated, the representation of the write operation does not even have to be a data flow - this is where data flows end, are not propagated any further (it does not even make sense to propagate them) and, therefore, we are going to distinguish between these two situations.

To make it easier to distinguish what we are talking about, let's give these three object types their names:

1. **Python Column Parent** is a data source which is being read. Data flow life cycle in a Python program starts here.

2. **Python Column** is the intermediate structure representing the data flow throughout its life cycle in the Python program.

3. **Python Resource Terminal** is the terminal node of a life cycle of a column. We want to avoid calling this *Column* terminal, because, at the end of the life cycle, the data flow being written does not necessarily have to be a column - it can be a hard-coded constant, a serialized class instance, or any other structure.

In Figure 6.3, you can see the expanded life cycle from Figure 6.2 translated to the newly named entities.

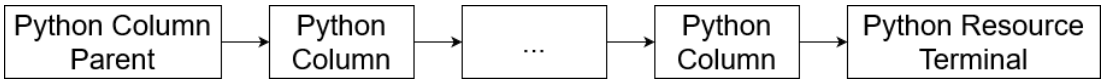

Figure 6.3: Data life cycle in terms of Python objects.

6.1.2 Column types

As mentioned in the previous section, there are three different types of columns that the MANTA platform distinguishes between - indexed, named and unknown. Because the output of our analysis is going to be transformed into a MANTA graph, we do not want to add any other type and will follow these three categories. Also, there is no need for a new column type, because if we cannot name a column by its name or position in a set of index, there is no other specifier how to disclose which column we are working with - it would be an *unknown* column.

However, we want to keep different information or implement different behavior for some types of columns - PySpark's DataFrame columns are able to perform different operations than database or file columns. Therefore, it makes more sense to have behavior of columns implemented in classes which extend column interfaces. Same goes for *column parents* and *resource terminals* - it is, obviously, going to be different when we work with a file resource than with a database resource. The core functionality is going to be the same (defined in an interface), but the implementation is going to differ. The exact functionality is going to be presented later in this chapter.

For columns, we are going to create three categorizing interfaces - indexed, named and unknown column. Every implementing class is going to define which

interface it is going to be able to represent. For file or database columns, it makes sense to only represent a single type, because it is never going to change, but for internal representation columns, for example, in Pandas or PySpark, where we want to process transformations and create new columns repeatedly, it is more convenient to not implement representation of columns per type.

Therefore, we won't have an unknown, indexed, named and indexed-and-named PySpark column, but we would have a single PySpark column which would be able to have any of the four different states, but it would provide functionality for every type (such as name or index retrieval). We talk more about the design of PySpark columns in Section 6.3.

6.1.3 Tracking origin

To be able to access all predecessors of a single column, we can simply provide an interface method which would return a direct predecessor (origin) of the column. From Figure 6.3 we can see that life cycles look like a linked list.

There can be situations when a column originates in two different columns, for example, in case of PySpark's `union`. In such cases, we can simply create two equal column instances representing the same DataFrame column, but with different origins. In the end, these columns are going to be visualized as one anyway, because MANTA Visualizer merges equal columns. However, because these instances are always going to be processed and propagated together, they will, eventually, be propagated to the same write operations.

6.1.4 Deduction

With chained columns, we can preserve information about column names and indexes quite easily. All we need to do is to preserve the order in which we gather information about columns. This means, that for example, when we have the following CSV-file-reading program:

```
1 with open('file.csv') as csv_file:
2     csv_reader = csv.reader(csv_file, delimiter=',')
3     for row in csv_reader:
4         print(row[1])
```

With a little simplification, we can generalize the processing of program into these steps:

1. We have got a file opened - store path `file.csv` to `csv_file` (line 1).
2. A reader is initialized and assigned to variable `csv_reader`. Create a new flow representing file read from the path in `csv_file` and this flow is the origin of a new unknown column, which is assigned to `csv_reader` (line 2).
3. `row = csv_reader` (line 3).
4. Process `row.__getitem__(1)` on line 4 as follows:
   (a) Find all indexed columns with `index == 1` and add them to the set of returned flows for this expression.

(b) Find all unknown file columns, for each of them create a new indexed file column with index 1 and origin being the unknown file column. Add the newly created indexed columns to the set of returned flows.

(c) Return the set of indexed file columns.

5. Process console write operation for all columns returned in previous step (line 4, again).

Therefore, we do not need to add any additional functionality to columns, column parents or resource terminals - previously outlined structure of flows suffices.

6.1.5 Creating resource columns

The final requirement we set during the analysis was a simple column creation. This is ensured very simply by turning `Python Column Parent` interface into a column factory. Using the abstract factory design pattern, we define three factory methods to be implemented by all column parents which would produce resource column objects: one for named, one for indexed and one for unknown columns.

Implementing classes, such as `FileReadFlow` or `ConsoleReadFlow`, decide which columns are created. For example, as we mentioned earlier in Section 5.1, the `ConsoleReadFlow` class would always return an unknown column, while the `FileReadFlow` class may want to return also named or indexed columns. Signatures of the methods can be seen in Figure 6.4.

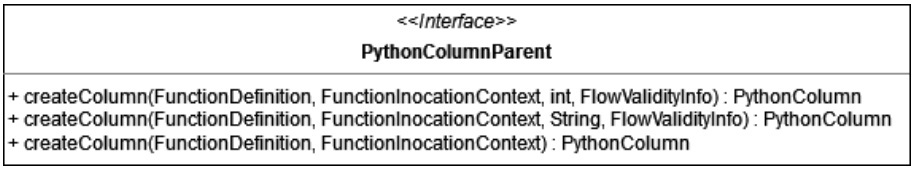

Figure 6.4: Signatures of the three methods to create a column in the interface `PythonColumnParent`.

6.1.6 Transforming columns to data lineage

The last thing we need to consider before we can finalize the design is to think about how is this chain of columns going to be transformed into a common intermediate structure, which is the base for the Intermediate Dataflow Generator component.

The generator only transforms input and output nodes and hides modifications inside the Python program. Therefore, we only need to correctly transform Python Column Parent, their most specific column, the Python Resource Terminal and its written column. Because the precision of column name or index is the most specific at the end of the column life cycle, it is easy to determine whether the written column is indexed, named, or unknown. We can simply look at the origin of the Python Resource Terminal and if it is a column, we resolve its type and get its name or index.

It is, however, more complicated to resolve which column is the source of data for the given written column in the Python Resource Terminal - without showing whence do the output columns get their data, the value of data lineage is not very high. We know that to get the input column, we must look among those columns which are *origins* of the output column. They form a chain of columns, as can be seen in, for example, Figure 6.3.

However, how many steps back from the output column do we have to take? We can outline several predicates which we will use to resolve as specific column name as possible, without losing correctness. When we think about how this can be done, we can use a decision diagram to define all constraints, as you can see in Figure 6.5.

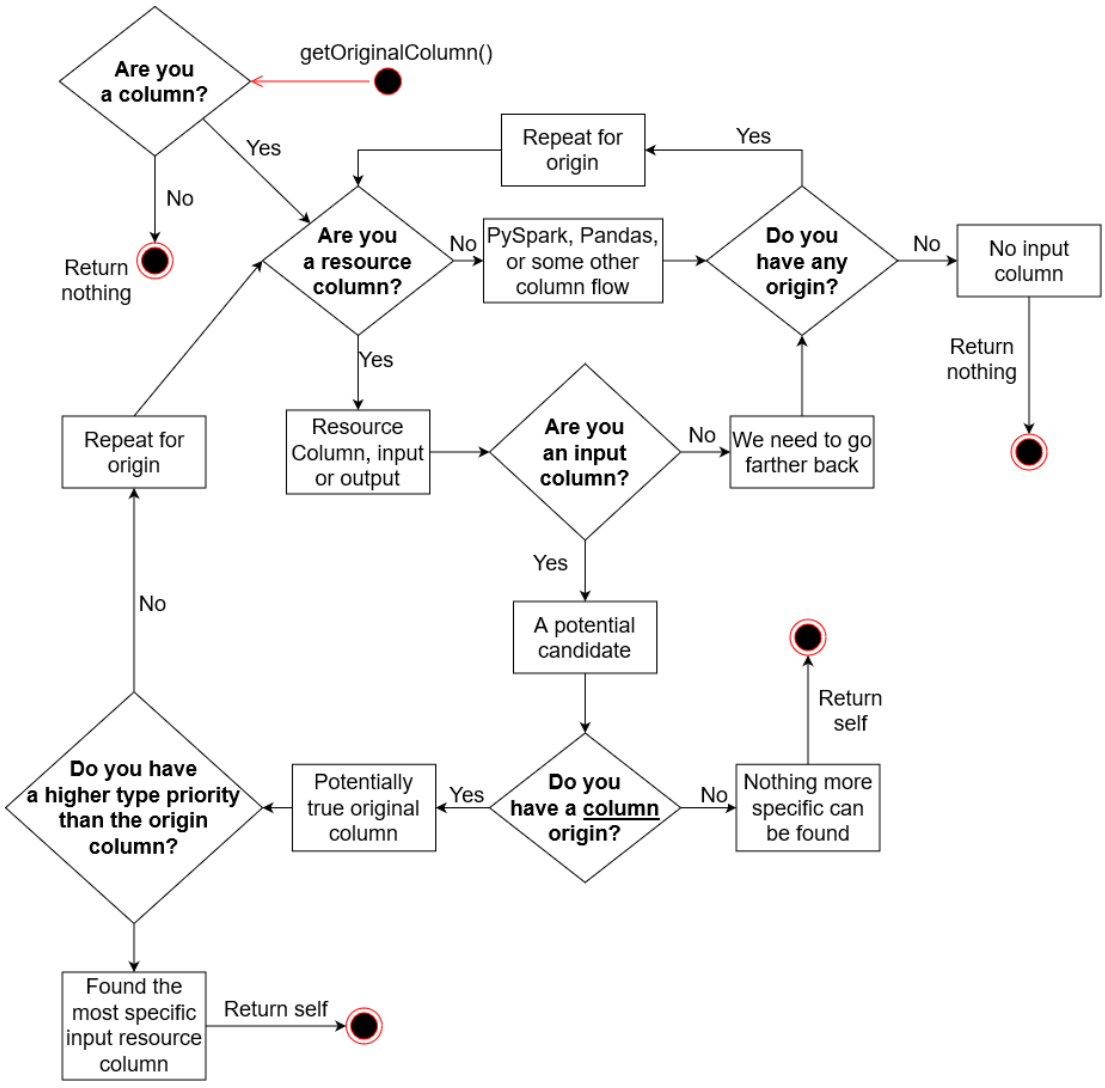

Figure 6.5: A decision diagram for getting the most specific input column.

We can call the method for retrieving the input column with the most specific identification as `getOriginalColumn()`. For every column, we must be able to determine:

- **Direction** - whether we are dealing with an input, output, or an unknown column. Unknown columns are cases when we are unable to identify which

role the column plays - usually in case of inner columns, such as PySpark or Pandas columns.

- **Type** - one of four types, as mentioned earlier: unknown, named, indexed, or indexed-and-named. These types must be comparable in the following manner: named, indexed, and indexed-and-named have got the same priority, unknown column has got a lower priority. Whenever we would go one step back in the column chain and this step would get us from a known column to an unknown column (within the same column type, e.g. input file column), we must prevent it to keep as much precision as possible.

To make it easier to think about how *input* columns would be resolved with this approach, you can see some examples in Figure 6.6. Examples are illustrated over file read and write operations as they are simpler to visualize, but database and console column resolution works analogously.

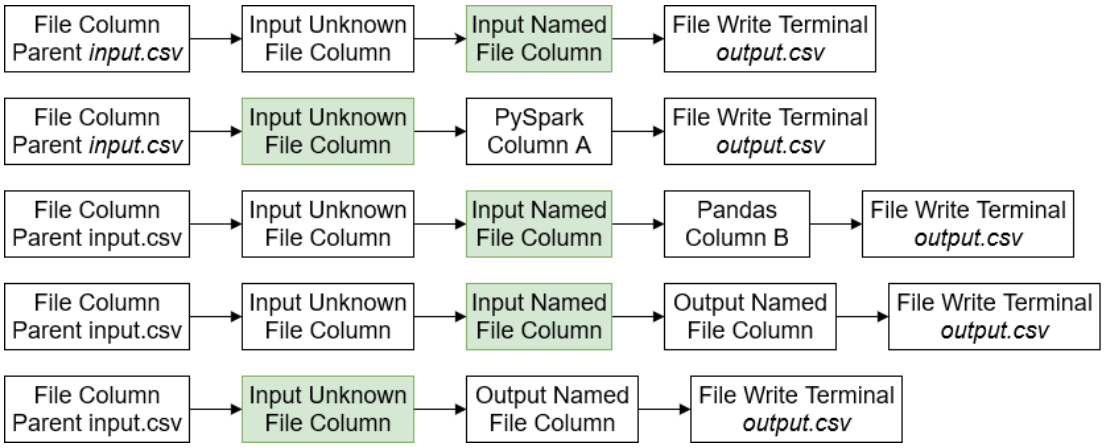

Figure 6.6: Resolution of the most specific input column. The resolved column is marked green.

In addition to resolving the *"original column"*, we also need to think about how to retrieve the `Python Column Parent` instance for the given column. This is a trivial task - we simply need to crawl back the column chain until we find a non-column origin. If this origin is a column parent, we return it, otherwise, we claim that there is no column parent (which implies that there are no input resource columns, since those *must* have a parent they belong to).

6.1.7 Design summary

Because the design of column handling was rather complex and introduced several different concepts, let us quickly summarize it, so that it is easier to understand all implications and properties that this design possesses.

The workflow of handling computed flows is going to change from resources (files, databases, console) to their columns. A standard life cycle of a flow, as depicted in Figure 6.7, is going to start at a resource flow, which represents reading the resource that provides data used in the Python program (interface `PythonColumnParent`).

This read flow is capable of creating relevant `PythonColumn` object belonging to this resource, such as a named, or an indexed column flow. The analysis then

uses these `PythonColumn` instances to propagate data flows across the program and compute the data lineage. Column propagation works in a layering manner, which means, that a new column is able to access all previous forms of its data `origin`.

Additionally, columns are able to tell their type, for example, an unknown, named, or an indexed column. They also contain information about their direction - *input*, *output*, or *unknown*.

At the end of the life cycle, when data flows are written somewhere (again, a file, a database, or the console), we register this write operation as a `PythonResourceTerminal` object for backwards transformation later, during the construction of the final data lineage graph.

When creating the lineage graph, it is possible to track the origin of write operations using the *origin* parameter of *PythonColumn* and *PythonResourceTerminal* objects. This effectively creates a backwards-linked list and using certain predicates, we are able to precisely determine the read resource columns which were used for the given written piece of data.

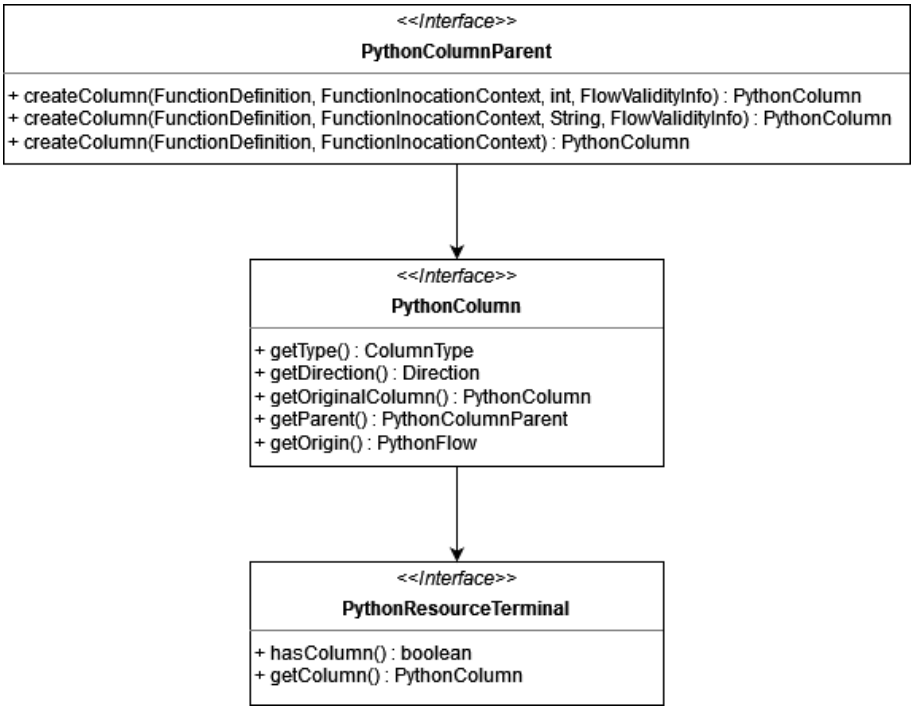

Figure 6.7: Column flow life cycle visualized by flow interfaces.

To make working with columns during the analysis more precise, we also introduced *trait interfaces*, as seen in Figure 6.8, which allow individual plugin propagation modes to filter out used columns more precisely, for example, when we are handling a `dataframe.select(*cols)` function in the PySpark plugin, we are definitely going to want to work with named and unknown columns, but definitely not indexed columns.

Thanks to this, we can ensure that columns filtered as *named* have always got a column name, which makes propagations easier to implement and work with.

The concept of columns is going to be important for the implementation of both CSV I/O operations and the PySpark plugin, see Sections 7.1.3 and 7.3.

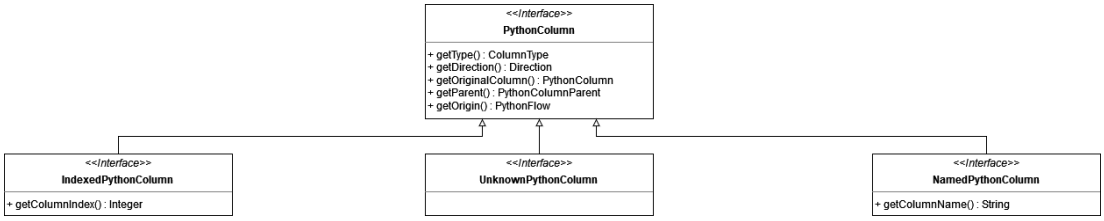

Figure 6.8: Trait interfaces for `PythonColumn` classes.

6.2 Flow Variables

Following the analysis of the Flow Variables feature in Section 5.3, we can proceed with the design of the feature. Before we introduce the proposal of the solution, let us first clarify certain details about how the scanner works internally, so that all new concepts are easily understandable.

Expression resolving

In the Python scanner, expressions are resolved in the `AssignmentFlowTarget #propagateAssignment()` method. There, expressions and flows related to them, stored in a map, are matched to the function invocation parameters. If an expression is not among keys of the map of expressions, a new `AExpressionFlow` object, representing the looked-up expression, with an empty set of related flows, is created and added. If it is present in the key set, all expression's flows are propagated into the flow set of the target expression (before the `foo()` invocation handling, `obj.field`'s flows would get propagated into the flow set of function parameter `self`).

For our use case, it would be ideal if we could modify this map in a way that would allow us also specifying 'inner' flows of the expression, so for expression `obj` we could also say that there is available expression `obj.field` and we could specify which flows represent it.

Expression resolution under the hood

Current resolution of flows per variable is very straightforward. In the analysis module of the Python scanner, identifiers are represented by two classes: the `VariableExpression`, representing a simple variable, such as `obj` or `my_var`, and the `IdentifierAccessExpression`, which are extensions to some other expression. This expression does not have to be an identifier, though. For example, in case of `foo().bar`, when `bar` is resolved against the result of the `foo()`, which is represented as a `ReturnExpression`. IdentifierAccessExpressions can be chained, creating a linked-list-like structure, as depicted in Figure 6.9.

Both `VariableExpression` and `IdentifierAccessExpression` extend the abstract class `IdentifierExpression`, as seen in Figure 6.10. They are going to be primary classes of interest in the design of a solution to this problem, however, other subclasses of the `AExpression` are going to be important as well, because we are also going to need to be able to analyze expressions, such as `foo().result.bar()`, where the base expression can be any other `AExpression` subclass.

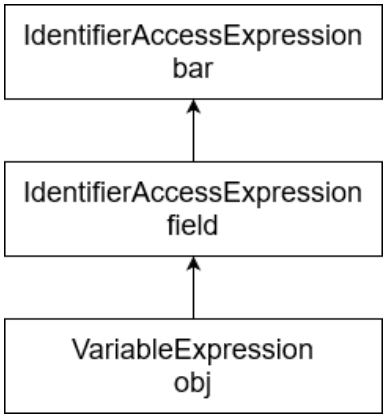

Figure 6.9: An example of how several `IdentifierAccessExpression` objects can be chained, in this case, for the representation of the expression `obj.field.bar`.

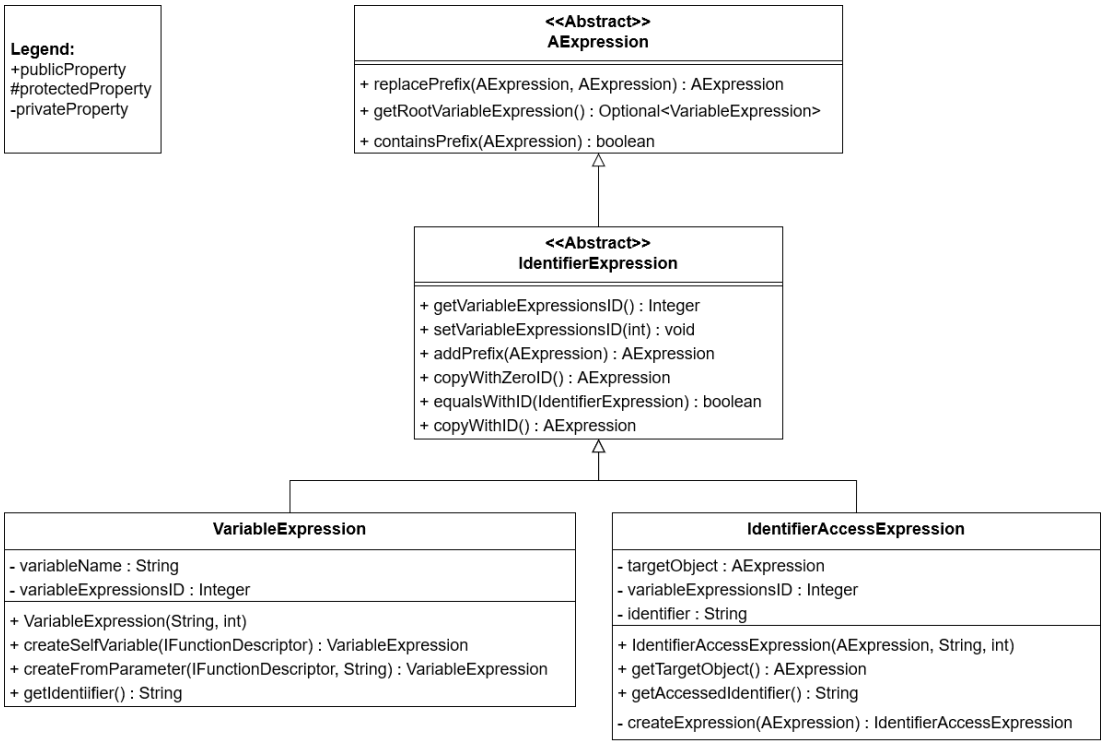

Figure 6.10: Class hierarchy of the `AExpression` class with focus on identifying expressions.

6.2.1 Inner flows

The first designed change is adding a new field to all flows named *inner flows*. An inner flow is, simply said, a definition of what should be returned if there is an inquiry about the flow of some field. If a flow represents a variable `foo` and has defined that upon requesting its field `bar` (as in `foo.bar`), it shall return constant *3*.

To store this information, we introduced the field `innerFlowMapping`, a map of `String` → `Function<String, Collection<PythonFlow>>`. Whenever a field is queried, the flow looks into this map and if there is an entry for the variable

name, it returns the result of applying the string parameter to the related function. In some cases, it can return itself or it can perform any other action with the information it has available. Thanks to this, there is little limitation as to what is defined in these inner flows, as long as it can be expressed in a function.

For creating an entry matching the `bar` example we mentioned at the beginning of this section, we would only need a single command:

```
addInnerFlowMapping("bar", key ->
    Set.of(new ConstantFlow(AExecutableFlow.START_NODE_INDEX,
    functionDefinition, 3)));
```

After adding this entry to the map, whenever any component asks the flow about the field `bar`, it returns a `ConstantFlow` object of value 3.

As you can see in Figure 6.11, we also introduced a new interface `PythonFlow` which is intended for simplification of the `AValueFlow` class, which grew quite big after adding the inner flow functionality, and it also allwos us to avoid wildcards in the code when we store the filtered-out `AValueFlow` into collections - instead of `Collection<? extends AValueFlow>`, `Collection<PythonFlow>` can be used.

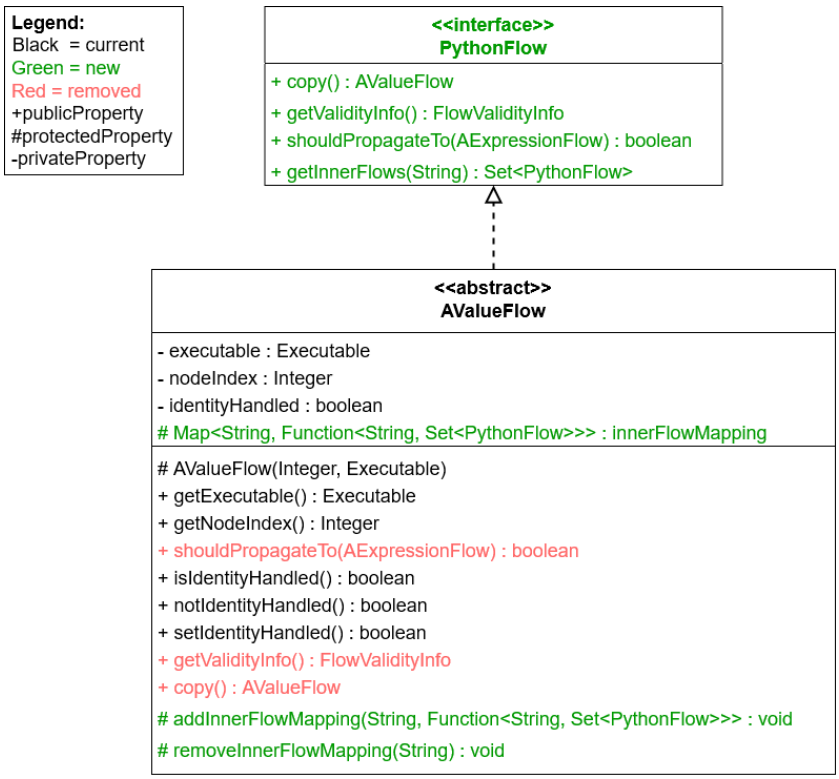

Figure 6.11: Changes to the `AValueFlow` class related to inner flows.

Adding this capability does not require any additional changes because the default behavior would be set in such manner that if the key is not found in the map of inner flows, an empty collection is returned. Therefore, unless we explicitly add some entries into the map, the analysis is going to work like before.

6.2.2 Expression resolving

The second part of the solution utilizes 'inner flows' we defined above. While tracked expressions are stored in a map used by the `AssignmentFlowTarget #propagateAssignment()` method, to avoid unnecessary increase in the map size, inner flows are kept as on-request flows. Whenever the analysis wants to resolve any `IdentifierAccessExpression` object, it is going to have to resolve the *identifier prefixes*. These prefixes are going to be collected by recursively going through the sought `IdentifierAccessExpression`'s prefixes until encountering a non-`IdentifierAccessExpression` object. At that point, the analysis looks for all flows of this non-`IdentifierAccessExpression` expression in the map of the `AssignmentFlowTarget` object and starts 'reconstructing' the sought `IdentifierAccessExpression` object by adding prefixes of chained expressions to the base - the `IdentifierAccessExpression` instance. Figure 6.12 illustrates proposed changes in the expression flows' lookup.

The algorithm of reconstructing these expressions is pretty much straightforward, as Figure 6.13 shows.

To better explain the idea, we can explain it on an example. Let us look for the right-hand-side expression in the following code fragment:

```
g.h = a.b[0].c.d.e.f
```

Iterating from the right to left, we determine the non-`IdentifierAccessExpression` `a.b[0]` (it is an invoke expression because the Python interpreter processes it as `a.b.__getitem__(0)` and so does the Python scanner).

Starting from this expression, there are four `IdentifierAccessExpression` objects which the scanner is going to need to process - `c`, `d`, `e`, and `f`.

The algorithm proceeds as follows:

1. `a.b[0]`
    - Check flows for `a.b[0]` flows
2. `a.b[0].c`
    - Query flows for `a.b[0].c`. For every flow from step 1, query for 'inner flow' `c` and add to the flow return collection.
3. `a.b[0].c.d`
    - Query flows for `a.b[0].c.d`. For every flow from step 2, query for 'inner flow' `d` and add to the flow return collection.
4. `a.b[0].c.d.e`
    - Query flows for `a.b[0].c.d.e`. For every flow from step 3, query for 'inner flow' `e` and add to the flow return collection.
5. `a.b[0].c.d.e.f`
    - Query flows for `a.b[0].c.d.e.f`. For every flow from step 4, query for 'inner flow' `f` and add to the flow return collection.
6. Return the output of step 5, or earlier if an empty set is a result of any of the previous steps.

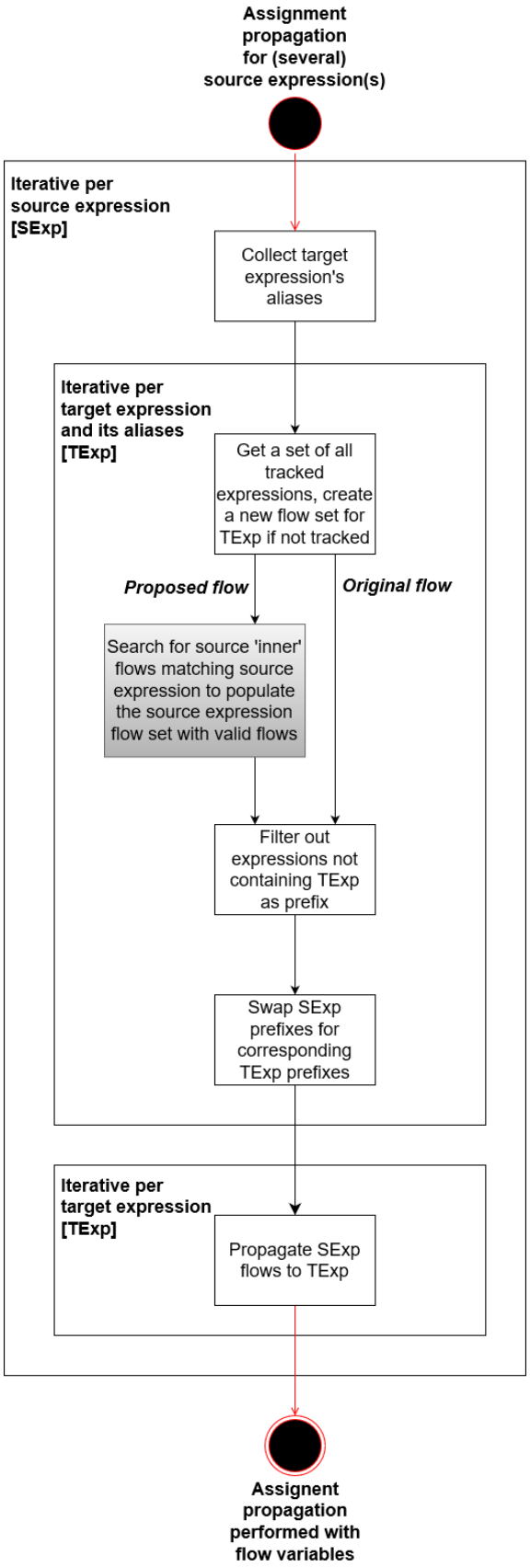

Figure 6.12: Proposed change in looking for flows related to an expression during the analysis (original and proposed solution).

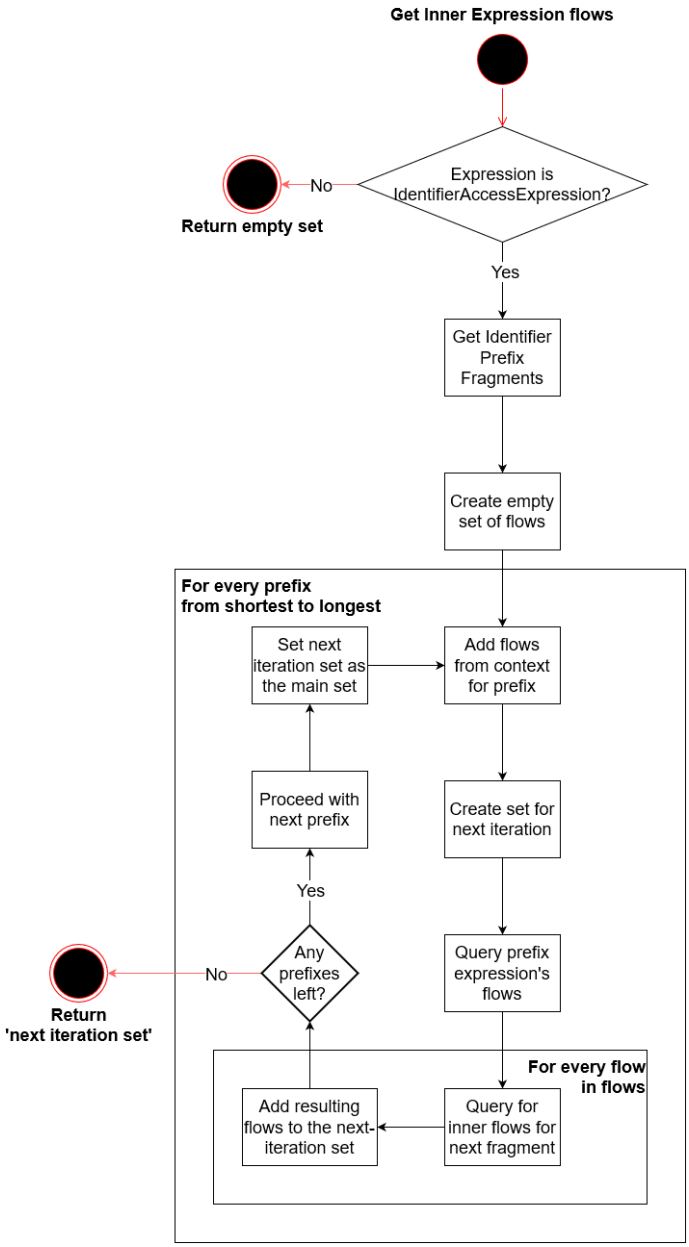

Figure 6.13: Algorithm for reconstructing sought expression by prefixes.

6.3 PySpark

In this section, we are going to describe the design of data structures that we are going to use for the implementation of the PySpark plugin. Because the plugin core of the Python scanner is already implemented, it is not necessary to design this part of the scanner and we can, therefore, only add more propagation modes and data flow types to handle PySpark functionality listed in Section 5.2.

6.3.1 DataFrame representation

The first and foremost thing we need to design is how PySpark's DataFrames are going to be represented. There are two options - representing DataFrame as a whole or only representing columns which would form DataFrames by simply

being assigned to the same variable in the analysis.

While the first option is very close to the actual representation in PySpark, the other option has got more advantages. If we look at the DataFrame class, it does not contain much information itself, we could say that it is rather a *management class* for its columns in the meaning that DataFrame offers functionality to users that they can use to manipulate columns and, therefore, also their wrapping structure - the DataFrame.

However, it is columns which represent the actual structure in the DataFrame and that are modified, rather than the DataFrame itself. Whether it is aliasing, joining or selecting columns, we simply modify the collection of columns we work with. This is no new point of view in the Python scanner as collections (sets, lists, or dicts) already work this way - there is no wrapping structure collecting all items together. Instead, they are represented separately as `CollectionItemFlow` objects and their affiliation to the same collection is implied by belonging to the same expression.

Another pro-column argument is that it makes more sense to us to work with columns since we have already designed a column-based representation of data flows in the scanner in Section 6.1 and it would not make sense to work with their wrapping data structures and making the whole solution more complicated and, logically, more error-prone.

Therefore, for reasons listed above, we decided to go with the column representation, from this point onward, we will reference them by their class name `PySparkColumnFlow`.

PySparkColumnFlow

Based on the intended supported functionality, we are going to need to store some data in the `PySparkColumnFlow` instances:

1. **Column name** - PySpark's columns can be named and, for some functions that we want to support, it is important to know names of the column. Therefore, this flow class is going to implement the `NamedPythonColumn` interface (see Section 6.1.7).

2. **Column index** - similarly to the column name, some functionality is also index-dependant. For this reason, the class will also implement the `IndexedPythonColumn` interface. In special cases, the `UnknownPythonColumn` insterface is going to be implemented - the column is unknown if neither name or index are known. However, name and index are not mutually exclusive - a column can have both the name and its index known.

3. **Table aliases** - names of the table the column belongs to.

4. **Known column aliases** - aliases of the column instance.

5. **Origin** - the flow that is the source of the data in the column. May be absent, for example, if an empty DataFrame column is created.

Because flows in the Python scanner are immutable data structures with no inner logic, only property getters shall be implemented by this class.

6.3.2 DataFrame schema

As explained in Section 3.4.1, a user may define a schema of a DataFrame, using classes `StructField` and `StructType` of PySpark. When creating a DataFrame with a pre-defined schema we have to realize that the structure of the new DataFrame is equal to the structure that user defined.

As already pointed out at the end of Section 5.2.1, the `StructField` class represents a single PySpark column in the new DataFrame and the `StructType` class represents the new DataFrame itself. The only change is in the origin of the data - while schema itself has no data origin, the DataFrame may have an origin.

Therefore, we only need to add origin to the schema, when it is present. This means that we do not need any additional data structures for classes `StructField` and `StructType` - they are going to be represented simply as `PySparkColumnFlow` class instances without any origin, like a mold for data.

6.3.3 DataFrame Reader and Writer

The next thing we have to think-through is the handling of I/O operations in PySpark.

It is important to notice that signatures of the reading and writing functions are almost identical. That means, that in most of cases, the algorithm for determining possible invocation targets would return both options - reading and writing. Because the reader and writer objects only contain *options* which can be present in both read and write objects, we are unable to determine the direction of data flow from this.

However, what we can do is following: if the code accesses a DataFrameReader instance `SparkSession.read`, a special object is returned, we will name it the `PySparkReaderFlow`, that is going to be present in the propagation mode handling the operation. Then, we can distinguish between input and output operations by simply trying to find the `PySparkReaderFlow` instance in the propagation mode's input flows - if it is present, we are dealing with a reader, otherwise we are dealing with a writer.

To make this work, we first have to be able to define the behavior of the `read` field access. To do this, we must have a flow class created for representing the Spark session, which we are going to call the `PySparkSessionFlow`, and then we can define the behavior for the `read` attribute, using the *Flow Variables* functionality we designed in Section 6.2. The `PySparkReaderFlow` class does not actually need to contain any information, its presence in a particular flow set is sufficient. This means that we can treat this flow as a *singleton* object.

For writer, we want to keep the information about DataFrame columns being written in the writer object. Because we do not have a writer object (write operation is determined by the absence of the `PySparkReaderFlow` instance, as we described earlier in this section), we have to implement a flow variable handling for the `write` field access in the `PySparkColumnFlow` instance. Returning columns themselves is going to be sufficient, because in the end, when processing the written DataFrame - we want to only work with the written columns and options of the writer.

6.3.4 Reader and writer options

The last thing we need to design is how to process options of the DataFrame reader and writer. Because options can contain any keys, not just those listed in the documentation, we have to create a data structure to store options in that would be flexible enough to contain any key-value pairs, but also the one that would not produce too much overapproximation by collecting all information in one place.

When processing one of the two *option-setting* functions, we simply need to take invocation arguments and transform them into a *flow* data structure which we can work with later. So, an obvious choice is to simply create a new type of flow, the `OptionFlow`, which stores the key and the value of the option, which can be used later in PySpark's propagation modes.

We must remember that neither keys nor values must be string values (they can be, for example, console inputs), so we can store them both as `PythonFlow` variables - the interface which is implemented by every flow present in the Python scanner.

Later, when using options, we simply have to look for all options and filter out those that matter to us. Some options, such as database *user* or *password* are not interesting for us in terms of data lineage. On the other hand, we are definitely going to want to use *format*, *path* or *dbtable* options.

By adding these options to the flow set of the invoked object (the dataframe reader or writer), we will have them available in propagation modes handling the loading and the writing of a DataFrame.

6.3.5 Example workflow

Now that we have got the key aspects of the PySpark plugin designed, to better explain how analyzing PySpark would work in relation to flow classes we described earlier in this section, we can explain what would happen in a simple PySpark program example from Figure 6.14, without mentioning unimportant steps not relevant to our purpose.

Line 1 - creates a new `PySparkSessionFlow` instance and stores it to variable `spark`

Line 2 - `spark.read` - the `PySparkSessionFlow` object returns the singleton object `PySparkReaderFlow`

Line 3 - both `DataFrameReader.csv()` and `DataFrameWriter.csv()` are analyzed, but only the `DataFrameReader.csv()` performs an action - a `PySparkReaderFlow` object is present. No schema is provided for the reading operation, so, first, a `FileReadFlow` is created for file *./input.csv*.

Then, its unknown `ResourceColumnFlow` instance is created, representing the output of the file read operation, This unknown column is the origin of the new unknown `PySparkColumnFlow` instance. It represents the data from the file stored in the DataFrame of variable `df`. We have no information about names or indexes of columns in the DataFrame, so we only create an unknown column.

Line 4 - accessing the `write` attribute returns all columns present. In this case, it is only the unknown `PySparkColumnFlow` object.

Line 5 - creates a new `OptionFlow` instance with key being `format` and its value `jdbc`. This flow is available in the `option` invocation on the next line,

```
1 spark = SparkSession.builder.getOrCreate()
2 df = spark.read \
3     .csv("./input.csv")
4 df.write \
5     .format("jdbc") \
6     .option("url", "jdbc:connection-string") \
7     .option("dbtable", "schema.sometable") \
8     .save()
```

Figure 6.14: A simple PySpark program using its reading and writing functionality.

together with the unknown column.

Line 6 - creates a new `OptionFlow` instance with key being `url` and its value `jdbc:connection-string`. This flow is available in the `option` invocation on the next line, together with the unknown column and the format option.

Line 7 - creates a new `OptionFlow` instance with key being `dbtable` and its value `schema.othertable`.

Line 8 - The save function does not find any `PySparkReaderFlow` object, so it processes its inputs. It looks for the format function, resolves it to be a database write operation. Collects connection details to the database - options *url* and *dbtable*.

With these details, the target table is known and the last step is to write all written columns there - in our case, it is just the unknown `PySparkColumnFlow` instance that was created on line 3.

You can see in Figure 6.15 how would the data lineage after the analysis look like for the program from Figure 6.14. During the transformation of this graph, the PySpark column would be concatenated and, therefore, a data flow from *./input.csv* to table *sometable* would be present in the MANTA data lineage graph.

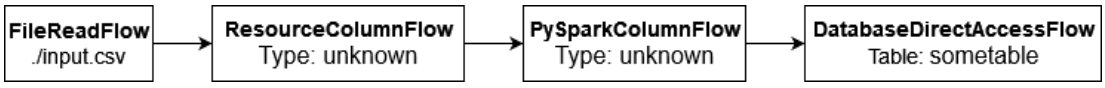

Figure 6.15: Data lineage of the example program from Figure 6.14.

6.4 Object-relational Mapping

In this section, we are going to focus on the design for the support of ORM tehcnology in SQLAlchemy, where mapping is defined in a declarative way. Since imperative way is more trivial, it does not require more thought and with a couple of simple changes, this approach can be supported.

In SQLAlchemy, both imperative and declarative approaches form equal structures, therefore, it shall be possible to achieve a common model for both approaches in the Python scanner as well. Because the scanner does not support variable descriptors and detailed complex class definition analysis, which are necessary for ORM analysis, the design does not go into too much detail, leaving

portions of design to be adjusted according to the implementation of these two essential requirements.

The first step to design the solution of the feature enabling the Python scanner to analyze ORM-based source code is to design suitable data structures which would allow the scanner to store all essential information found in the source code and to let developers work with these data structures effortlessly to avoid potential bugs in the code. From the analysis, it can be understood that the information needed to be stored in the designed data structures is:

1. Relations between declared ORM classes and the name of the table they are mapped to.

2. Columns of every mapping class - names of the column in both class and the table (function `mapped_column(...)` allows to define the name of the table's column being mapped, this name does not have to be the same as the name of the variable to which this column instance is assigned).

3. Information about relationships between classes, as we discussed in Section 4.5.1.

The mapped class declaration can, with a little bit of abstraction, be considered similar to PySpark and its schemas. They both define the mould for data, where it should fit. In case of PySpark, the form for the data is defined via a schema, while in case of SQLAlchemy's ORM and its declarative approach, it is defined via mapping classes (using assignments or type annotations).

Therefore, the most obvious way to store information about columns is to have a similar data structure to the `PySparkColumnFlow` class, which has its own metadata (such as a name, an index, or aliases), which are already present when the column is only defined in a schema, but it can also contain data, in form of its origin - like all other column flows.

The same would, presumably, be desired for ORM columns. Both PySpark columns and ORM columns can contain type information, but that is not important information for us. The flow object defining an ORM column shall, therefore, contain following information:

- Name of the table to which it is mapped (mapped class' `__tablename__` field value).
- Column's name in the table.
- Column's name in the class (variable name).
- Origin (nullable when declared as a mapped class, may contain value when mapped class is instantiated).

For the case of relationships, it is needed to keep even less information. According to the configuration of the relationship, only a few configurable fields are needed to be tracked:

- Name of the field within its class that contains the relationship (name of the variable in the class).

- Name of the class which can be accessed via the class variable (the related class).
- Flag whether the field returns a class instance or a collection of class instances.
  - For declaration `children: Mapped[List["Child"]] = relationship(back_populates="parent")`, a collection (List) of the `Child` class instances is returned.
  - For declaration `child: Mapped[Child] = relationship(back_populates="parent")`, one object is returned.
  - This affects whether the scanner should work with the object directly, or whether it shall work with a collection of objects.

It is, however, impossible to link classes together at class initialization in certain situations, for example, when two classes related with a foreign key allow to access each other:

```
class User(Base):
    ...
    addresses: Mapped[List["Address"]] = \
                    relationship(back_populates="user")


class Address(Base):
    ...
    user: Mapped[User] = relationship(back_populates="addresses")
```

Classes are, logically, processed in the exact order in which they appear in the source code and, in this case, it is guaranteed that the scanner would not know about the related classes, which is declared after the currently processed class (in the case above, the `Address` class referenced on line 3 would be yet unknown to the scanner).

Therefore, the scanner must only store information about the name of the referenced class and perform lookup upon the request. That is, when the `addresses` field is accessed, the scanner looks for the class referenced by this variable, in this case, the Address class. Since class and function definitions are processed sequentially, the same way that Python does it, unless a special situation happens, the scanner will know about the `Address` class by the time the `addresses` variable is accessed.

It is necessary to keep this information in a data structure which would be mutable. It is not possible to say upfront when the modifications in the ORM mappings are over and, therefore, an immutable option is not possible. Such situation can happen when there is a mapped class altered at runtime. It is especially unpleasant in the case of imperative ORM mapping approach, as you can see in the code snippet in Figure 6.16.

Unless the last command on line 11 is invoked, the scanner has no idea that the class `User` would ever represent a mapping class and even if it did, there would be no information about its mapping metadata unless it is paired with the

```
1  user_table = Table(
2      "user",
3      mapper_registry.metadata,
4      Column("id", Integer, primary_key=True),
5      Column("name", String(50))
6  )
7
8  class User:
9      pass
10
11 mapper_registry.map_imperatively(User, user_table)
```

Figure 6.16: Source code of a program which changes the ORM class model during runtime.

`user_table`. This means that to create an immutable data structure, the scanner would first have to analyze all the source code, only looking for the ORM-related source code, to establish the model upon which the ORM analysis would operate, and after that, another analysis of the source code with immutable model would be run. This is an unnecessarily complex solution which would add non-trivial time complexity to the scanner's analysis with very little benefit.

Instead, an auxiliary structure could be used to progressively update and aggregate information about the registry and its related mapping classes. We can name the mutable structure analogously to the SQLAlchemy's ORM `registry` - the `ORMRegistry`. Both classes serve to resolve columns and relationships and this way it is the best to keep the naming similar. Other ORM-related objects would only contain the reference to the `ORMRegistry` instance, which would provide up-to-date information without a need to modify flows themselves, which should remain immutable.

This `ORMRegistry` class should, then, be able to map:

- An ORM table to its ORM columns.
- ORM table variables to other ORM tables mapped via a relationship.
- Name of an ORM mapping class to a table name - in some cases, only the table name is known, while in other cases, we only know the name of the ORM mapping class.

Additionally, it should be possible to query this information when needed, even if the registry state is not yet final.

Lastly, it is important to consider how classes will be represented. In order to separate functionality, it seems to be the best idea to not directly connect classes with their fields because of the issue with relationships. In some cases, the class would be able to access its columns, but if a relationship-specified class was accessed, the ORM registry would have to be used. In order to keep related functionality together, only representing an ORM mapping class as a simple proxy between the scanner (or a propagation mode) and the ORM registry is reasonable.

If any object was needed, whether it is a column or a related class, the ORM registry could be used.

The ORM mapping class would not contain any logic, it would only be used to get to the related mutable object. All information that is needed for the class representative to know is a reference to the related ORM registry and the name of the class, or the name of the table, that it represents. These values can be altered as they are both considered to be static in the program and it is only a matter of implementation which value is used in mappings and lookup method invocations.

With these classes outlined, we can give names to individual classes of the ORM model of the Python scanner. The class representing a declared column could be named the `ORMMappingColumnFlow`, the proxy class for representing a mapping class would be named `ORMMappingClassFlow`, the entity acting as a placeholder for a relationship mapping will be the `RelationshipMapping`. You can see classes, their relations, and fields in Figure 6.17.

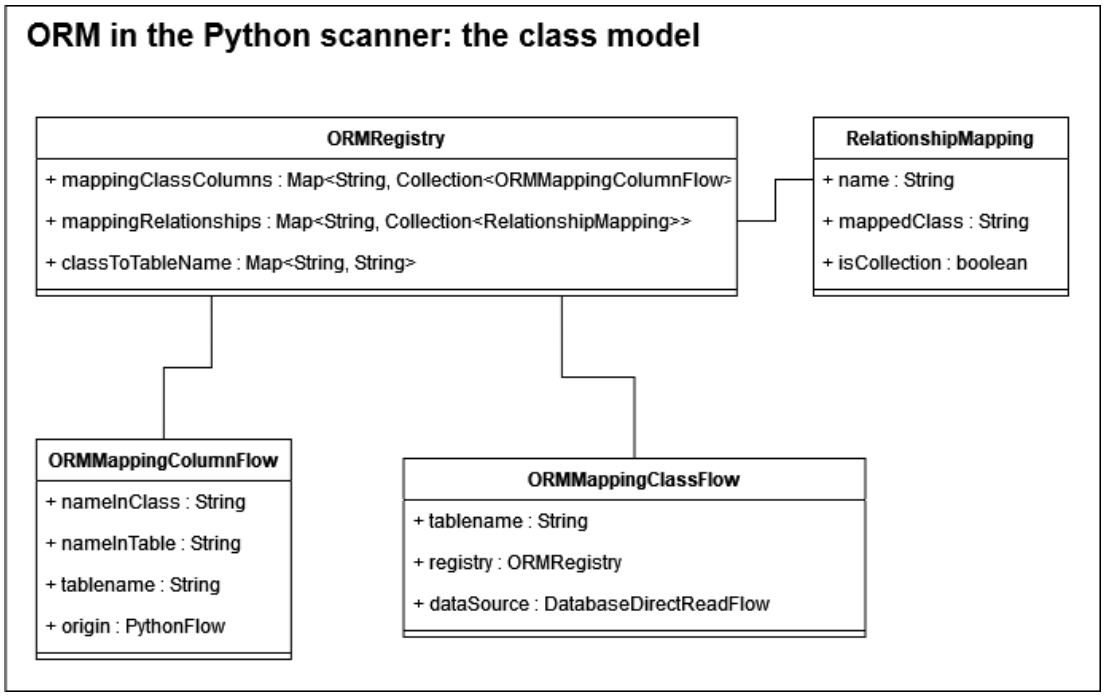

Figure 6.17: Class model of the designed ORM-analyzing feature.

So far, the design has been a little abstract and it is better to explain it in examples. In the following sections, we will explain the usage of these classes on concrete examples of initialization and behavior in various operations. Note that the DELETE operation is omitted as it produces no data lineage information currently relevant for the Python scanner.

6.4.1 Initialization of mapping defined declaratively

The first step to supporting the ORM technology in the scanner is enabling it to process the metadata information defined in mapping classes. With ORM model defined above, we can explain how they would be used in the case of a simple

use case with two mapping classes named `User` and `Address`. These classes will, then, be used in code fragments for concrete ORM operations below.

In the code snippet below, there is a class which defines the declarative base class for the ORM of the program. Then, there are two more classes, the `User` and the `Address`, which represent two database tables, `user_account` and `address` respectively. In SQLAlchemy, every instance of these classes represents a single row in their related table. As mentioned earlier, they define the form into which table data is loaded. It is necessary to, therefore, analyze how these classes are defined.

The first step is to map the `Base` class to an instance of the `ORMRegistry` class. Every `Base`-extending class uses the same registry implicitly. If another class extended the `DeclarativeBase` class, another instance of the registry would have to be used.

Once the registry for the selected base is created, it should be clear for all extending classes which is their parent registry - this information can be simply deduced from the base class and the registry instance assigned to it. What the scanner needs to do when initializing a new class definition object is to analyze all of its variables. There are, essentially, four types of variables that can be encountered:

- `__tablename__` defining the name of the table which it maps (within a given `Session` instance).
- Table column fields which represent columns of the mapped table.
- Relationship fields allowing users to access instances of related table rows (other ORM classes).
- Other fields which are not mapped to columns and only serve the implementation.

The first three categories are interesting for data lineage analysis, the last one is not. From invocation of functions `mapped_column` and `relationship`, it is quite easily possible to determine which columns are mapped and which classes are related to each other.

When it comes to the analysis of fields declared using PEP 484 [16], this information can be retrieved by parsing the type hint. Of course, this has got certain limitations, for example, if the `int` return type is defined in a non-standard way, for example, as a concatenation of three characters: `def foo() -> chr(73) + chr(78) + chr(84)`, instead of the standard `def foo() -> int`. Both values are allowed, but the Python scanner may ignore non-standard ways to define type hints as even IDEs only use identifier-defined type hints and do not evaluate expressions.

```
1  # ...relevant imports...
2
3  class Base(DeclarativeBase):
4      pass
5
6
```

```
7  class User(Base):
8      __tablename__ = "user_account"
9
10     id: Mapped[int] = mapped_column(primary_key=True)
11     name: Mapped[str] = mapped_column(String(30))
12     fullname: Mapped[Optional[str]]
13
14     addresses: Mapped[List["Address"]] = \
15                   relationship(back_populates="user")
16
17
18 class Address(Base):
19     __tablename__ = "address"
20
21     id: Mapped[int] = mapped_column(primary_key=True)
22     email_address: Mapped[str]
23     user_id = mapped_column(ForeignKey("user_account.id"))
24
25     user: Mapped[User] = \
26              relationship(back_populates="addresses")
27
28
29 engine = create_engine(
30               'mysql+pymysql://root:admin@localhost/schema')
31 session = Session(engine)
```

When the scanner approaches line 7 in the code snippet above, it first determines that the `Base` class is a declarative base and, therefore, it should be able to retrieve its `ORMRegistry` instance. Then, it processes and categorizes class fields.

The `__tablename__` field is present at most once, other types are either ignored (*Other fields*) or transformed into `ORMMappingColumnFlow` and `RelationshipMapping` instances. Once all information available in the source code is collected and categorized, the class shall register itself in the registry, so that it can be worked with later. The whole process can be seen in Figure 6.18.

If all necessary steps for the proper initialization of mapping classes are followed, the state of the registry at the end of the process should be approximately as it is visualized in Figure 6.19.

We are going to consider this state as the base for the ORM operations described later in this chapter. Identifier `User` contains an instance of the `ORMMappingClassFlow` class, referencing the registry and containing information about the class it represents.

Again, as described above, this can be either the table name or the class name - this is the implementation detail that needs to determine which identifying value will be used in registry's maps. It does not affect the functionality or this design, though.

On the last two lines, it can be seen that an engine and a session are initialized, which means that the session is able to identify the data source which is used for searching mapped tables.

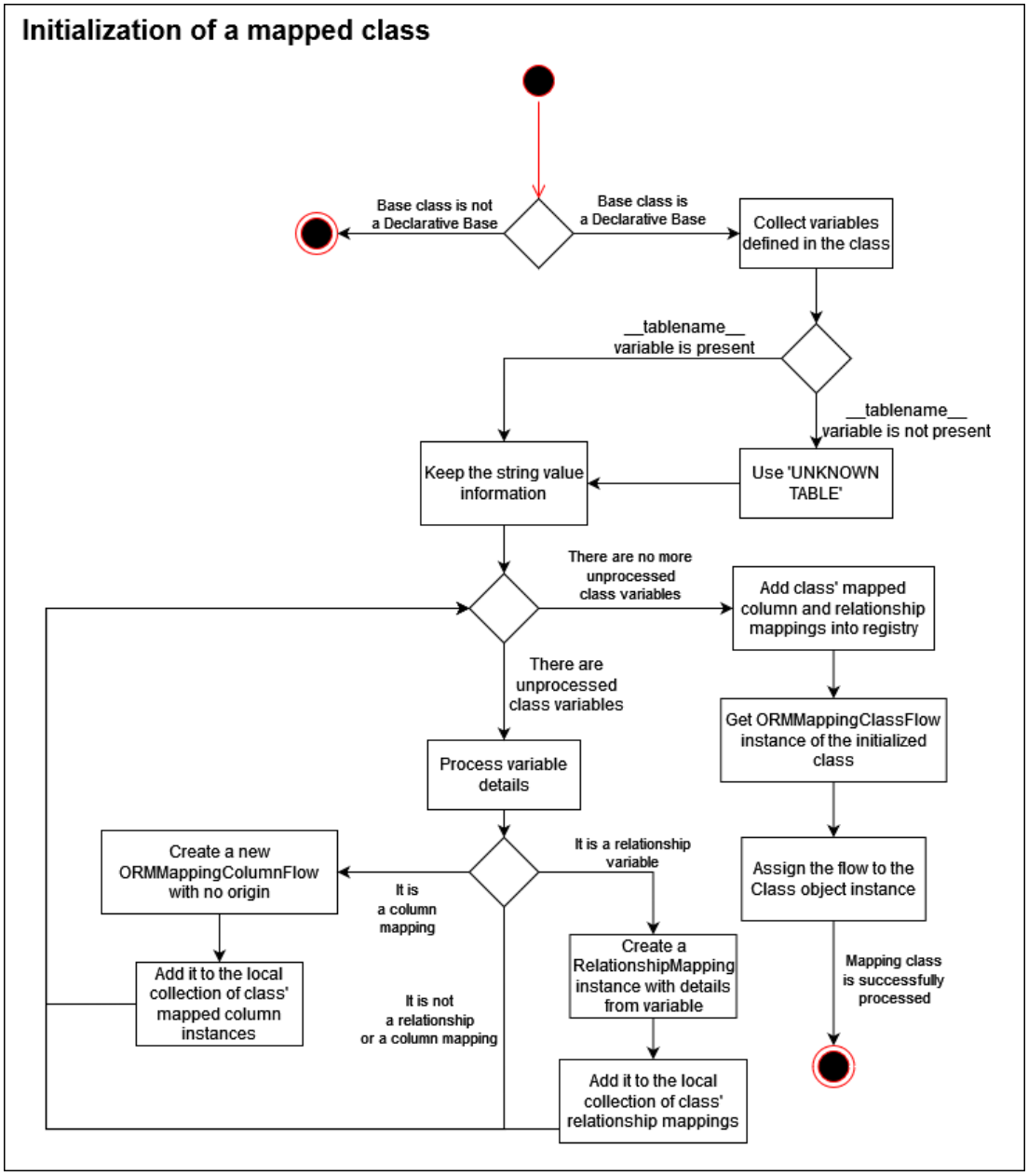

Figure 6.18: Steps of the algorithm to initialize mapped ORM classes.

6.4.2 Accessing columns

Because in case of mapped classes only a proxy class is returned instead of all columns of the class, it is necessary to determine how columns would be accessed. For example, it is a standard practice within the ORM technology to modify mapped class instances by field assignment in applications:

```
user_instance = ...load a row from the user_account table...
user_instance.fullname = 'changed-value'
```

On line 1, only an `ORMMappingClassFlow` instance of the `User` class is stored into the `user_instance` variable. However, on line 2, its column needs to be accessed. A simple solution is possible using the *flow variables* feature. Whenever a field of `user_instance` (or any other `ORMMappingClassFlow` object) is queried,

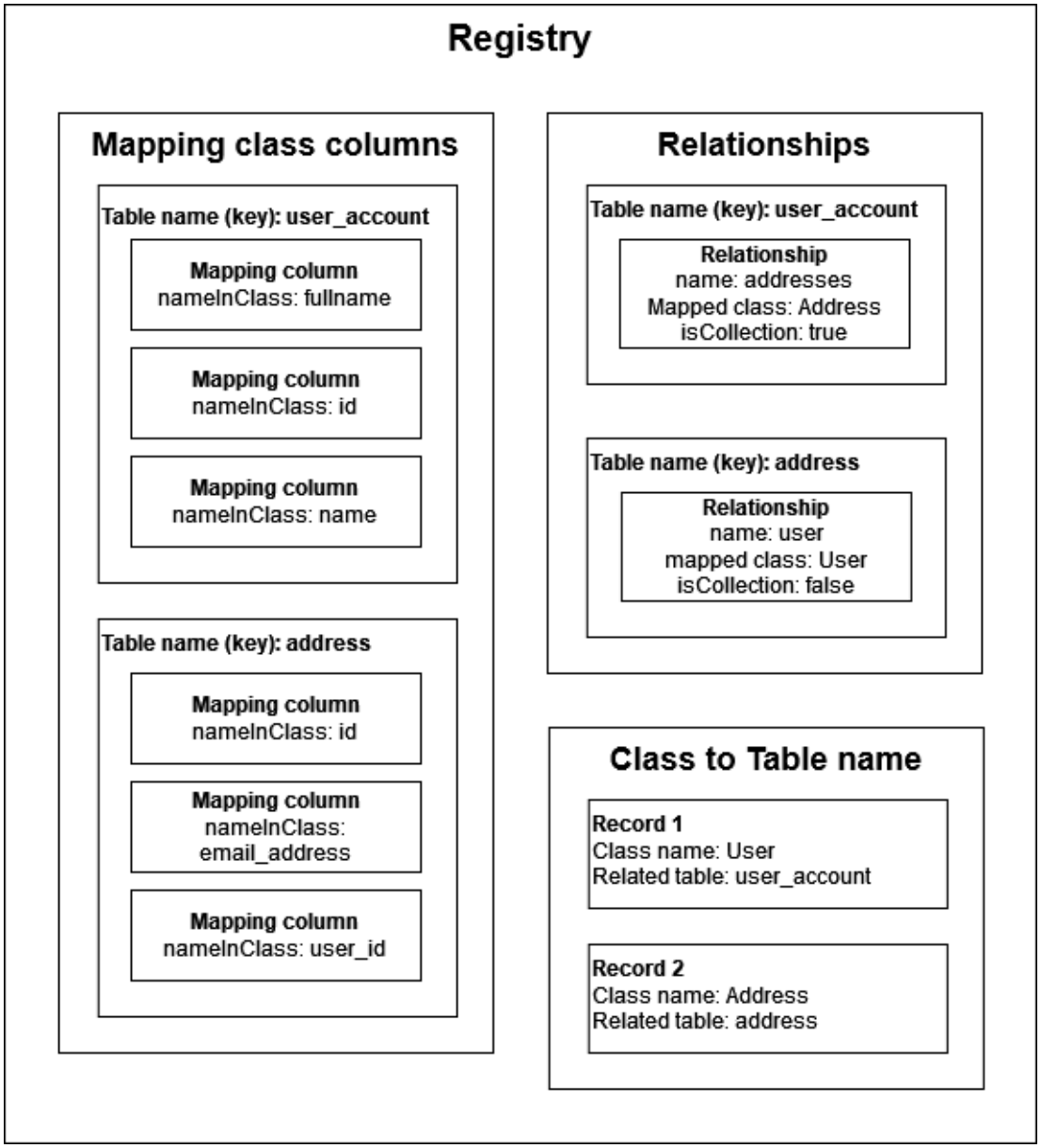

Figure 6.19: State of the ORMRegistry instance after processing the two ORM mapping classes' initialization.

the flow only needs to forward the query to its registry. The registry may return a column, a class (defined by a relationship), or it may return some default value - this is an implementation detail to be specified later. The `ORMMappingClassFlow` object does not need to know all the information and its structure upfront, only a simple delegation to the registry is needed.

However, one more thing is necessary to be stored. When an `ORMMappingClassFlow` instance is assigned to the `User` variable during the mapping class initialization, no data source is present as it is only a mould for the data.

However, when it is initialized via some session, then it becomes an object, instead of the 'mould' and the origin of its data needs to be stored somewhere. The `Session` instance knows the connection string of the engine it is related to and, therefore, if a `Session` object creates an `ORMMappingClassFlow` object, it

needs to provide the class flow with this information, so that the columns received from the registry can be cloned with a correct data origin.

In general, the two-line code snippet above would be analyzed by the Python scanner in the following steps:

1. Right side of the assignment on line 1: creates an `ORMMappingClassFlow` instance of the relevant class by cloning the mould of the `ORMMappingClassFlow` and providing it with the connection string retrieved from the session object.

2. The new `ORMMappingClassFlow` instance stores the connection string as a `DatabaseDirectReadFlow` object.

3. The new `ORMMappingClassFlow` instance is assigned to the `user_instance` variable.

4. Line 2 - resolving left-hand side of the assignment: query for the `fullname` flow variable over `user_instance`.

5. The `ORMMappingClassFlow` object queries its related `ORMRegistry` instance for flows related to the class `User` with name `fullname`.

6. The `ORMRegistry` instance returns a single `ORMMapingColumnFlow` object.

7. The `ORMMappingClassFlow` instance copies the returned column flow and add an origin to it - a named resource column flow with the parent being the class flow's `dataSource` and the column's name being `fullname`.

8. This new column flow is assigned to the `user_instance.fullname` expression's flow set, but also to the `user_instance`, because we may insert this variable to the session (see INSERT operation below), not just the `user_instance.fullname` and if we did not do this, we may lose the information of the assigned field value.

9. The string value `changed-value` is assigned to that expression's flow set as well.

The situation works the same way for the case when the `ORMMappingClassFlow` object is returned (when related class is accessed, not a class field).

6.4.3 INSERT operations

For the case of INSERT operations, there are two general approaches that can be used - individual and bulk insertions. We will explain how both approaches would work in current design.

Standard INSERT operation

The code below performs a single row insertion into the `user_account` table. It uses the default initializer defined in the `DeclarativeBase` class which uses arbitrary keyword arguments - `__init__(**kwargs)`. The instance is, then, assigned to the variable `user`. On line 2, the instance is added into the session,

which keeps this information internally and on the last line, the transaction is committed, which means that the changes have been projected into the database table.

```
user = User(name="sandy", fullname="Sandy Cheeks2")
session.add(user)
session.commit()
```

The first thing that needs to be thought about is when are the changes considered as definite? When is the transaction seen as completed? One, obvious way is to wait for the `commit()` function invocation, which would prevent incorrect deduction in case a `rollback()` function was called instead. However, this has got two problems:

1. An operation with calling `rollback()` at the end makes no sense, except for the try-catch or if-else blocks, for example, when there is some problem during the transaction commit and the whole transaction needs to be rolled back (but in some possible scenario of the program, the `commit()` function would be invoked anyway. This means that it is not so crucial to wait for either a `commit()` or a `rollback()` operation.

2. There may be a committed transaction even without the explicit `commit()` or `rollback()` operations. For example, when the session is created and auto-closed in a `with` statement:

   ```
   with Session(engine) as session:
       user = User(name="sandy", fullname="Sandy Cheeks2")
       session.add(user)
   ```

Additionally, the *autoflush* feature automatically commits all changes upon a query operation (see Section 4.5.1 for more details), this feature is enabled by default.

Because of these two problems, it seems that considering any data changes in ORM as effectively final may be a better idea. This means that for the INSERT operation, adding user into the session may already by considered as a committed change. Taking this into consideration, the Python scanner could ignore line 3, performing only following steps:

1. Process constructor, launch propagation mode that processes initialization function of the `DeclarativeBase` class with arbitrary keyword arguments. Map arguments to the `User` class' columns (using keyword arguments' keys). Copy these matched columns, setting their origin to be the argument value.

2. Store the newly created column flows into the variable `user`.

3. Continue with line 2.

4. Launch the propagation mode of the function `Session.add()` , where for every added column flow (stored in the `user` variable, passed as a function argument), create a new `DatabaseDirectWriteFlow` instance, retrieving connection string from the `session`, and table and column names from the `ORMMappedColumnFlow` object. The flow being written (the origin) would be the `ORMMappedColumnFlow` instance itself.

Bulk INSERT operation

For the bulk INSERT, the situation is quite similar. Let us have the following code:

```
session.execute(
    insert(User),
    [
        {"name": "steven", "fullname": "Steven Chen"},
        {"name": input(), "fullname": "James Bond"},
        {"name": "carlos", "fullname": "Carlos Rodriguez"},
        {"name": "stewart", "fullname": input()},
        {"name": "juliet", "fullname": "Juliet Capulet"},
    ],
)
session.commit()
```

In this case, it is important to notice that a bulk insert is, essentially, the standard INSERT concentrated in a single invocation of the `execute()` function. Therefore, expected steps for its successful analysis are:

1. Again, ignore the `commit()` invocation on the last line.
2. Process `insert()` function on line 2. Return the `ORMMappedClassFlow` object of the `User` class. Return also the information that the INSERT operation was used (either as a separate flow instance or in a wrapper together with the `ORMMappedClassFlow` instance) - not important for the design.
3. Process the `execute()` operation.
4. From the presence of the `ORMMappedClassFlow` instance, determine that the inserted class is `User`. In some corner cases, even several `ORMMappedClassFlow` instances may be present. In such case, for each of them, perform the remaining steps.
5. For every dictionary in the second argument of the `execute()` function, behave as if an individual INSERT happened. Keywords in `DeclarativeBase.__init__(**kwargs)` are passed as a dictionary as well, so no big changes are necessary.
6. Because there is information present about an insertion operation, also simulate the `Session.add()` operation using the output of the previous step.

The result would be the same as if there were five individual INSERT operations executed.

6.4.4 SELECT operations

In case of the SELECT operation, the scanner must be able to determine which classes (or even columns) are being selected. Let us have the following code:

```
users = session.execute(select(User))
```

Again, the `Session.execute()` function is used, so the output of the invocation of `select(User)` must provide it with the information that a SELECT operation is performed and which objects are being selected (classes or columns). The steps of the scanner should be:

1. Analyze the `select(User)` invocation.
2. Propagate the `User` class' `ORMMappedClassFlow` instance and the information that the SELECT operation was used.
3. Analyze the `Session.execute()` function.
4. Determine that a SELECT operation is being executed.
5. For every `DatabaseConnectionStringFlow` object in the `Session` instance, create a new `DatabaseDirectReadFlow` instance.
6. For every selected `ORMMappedClassFlow` object:
   (a) For every created `DatabaseDirectReadFlow` object:
      i. Create a copy of the `ORMMappedClassFlow` instance, setting the `dataSource` to be the `DatabaseDirectReadFlow` object.
7. For every selected `ORMMappedColumnFlow` object:
   (a) For every created `DatabaseDirectReadFlow` object:
      i. Create a new `ResourceColumnFlow` instance named equally as the iterated `ORMMappedColumnFlow` object (if the name is unknown, create an unknown resource column flow) using the `DatabaseDirectReadFlow` object as its parent flow.
      ii. Create a copy of the `ORMMappedClassFlow` object with its `origin` being the create resource column flow.
8. Wrap the newly created mapped column and class flows into some kind of a wrapper flow class, so that propagation modes can be created for iterating this `Result` instance (see details in Section 4.5).
9. Return the wrapper flow class.
10. Assign returned wrapper flows to the `users` variable.

In case of the code snippet above, step 7 would be skipped as no columns are being selected. However, the following source code would make use of this step, instead of the step 6:

```
user_details = session.execute(select(User.name, User.fullname))
```

6.4.5 UPDATE operations

The UPDATE operation is the most complex as it is not handled entirely just by function invocations. While in case of the INSERT and SELECT operations, explicit functions were invoked performing this operation, in case of the UPDATE operation, as can be seen in the code snippet below, the UPDATE operation is done simply by updating a field value of a mapped class instance and committing these changes.

This means that the analysis of this operation needs to be different. In the code below, first, all records from the table `user_account` with ID equal to 2 (exactly one record as column ID is the primary key) are selected and, then, the `scalar_one()` function returns the single record present in the `Result` object. This object is assigned to the variable `sandy` and its column `fullname` is changed to value provided by the user via the console input. This change is, then, committed by the session.

As described in the analysis, the `Session` instance maintains the list of 'dirty' mapped objects and commits changes (updates) only those records, which are dirty. In this case, only the table cell in the row with `User.id == 2` and column `fullname` would be updated.

```
sandy = session.execute(select(User) \
            .where(User.id == 2)).scalar_one()
sandy.fullname = input("Enter user's full name: ")
session.commit()
```

We discussed earlier that it is not preferred to wait for the `commit()` function invocation which may never happen. If we, again, modify the code to use the with-statement, the `commit()` is invoked implicitly:

```
# equal to the code snippet above
with Session(engine) as session:
    sandy = session.execute(select(User) \
                .where(User.id == 2)).scalar_one()
    sandy.fullname = input("Enter user's full name: ")
```

An option to analyze this operation is to modify the way in which assignments work. The analysis could keep information whether the set of flows assigned to a variable (or any expression) contains an instance of the `ORMMappedColumnFlow` class with origin being a resource column flow of a database.

This signs that the column's data was loaded from a table, for example, using the SELECT ORM operation and it is mapped to it. This is very easy to check as there are only two ways how new flows can be added into a flow set instance (more specifically, into its modifiable version named `MPythonFlowSet`):

1. Adding the flow into the constructor.
2. Invoking the `MPythonFlowSet#add(PythonFlow)` method.

In both cases, it is relatively simple to verify that such flow occurs and if it does, the set needs to keep all these flows in a separate collection - they would

be used later. Flows may never be removed from a flow set, so they may never be removed from the list of found 'special' flows as well.

When an assignment occurs, the analysis only needs to check that the flow being assigned to a variable's flow set contains at least one of the 'special' `ORMMappedColumnFlows` instances. If that is the case, a new `DatabaseDirectWriteFlow` object needs to be created for each of these `ORMMappedColumnFlow` objects. The origin of the data is going to be the assigned expression (the right side of the assignment), in our case the console input from line 2, and the connection details can be retrieved from the origin of the `ORMMappedColumnFlow` object (its resource column flow is a database, which means that it contains connection details - the table name and the connection string).

This operation has, however, got a corner case when the data lineage may be incorrect, for example when the assignment of some data is to simply 'clean up' values from the database, and these changed objects are not written back into the same table, but, for example, to a file or other data source. In the example below, a console output is shown as the destination for the cleansed data to make the code more simple:

```
sandy = session.execute(select(User) \
            .where(User.id == 2)).scalar_one()
if sandy.fullname is None:
  sandy.fullname = input("Enter backup value: ");
print(sandy)
```

This is one of the cases when a simpler and faster solution would be traded for a potentially incorrect data lineage, but these cases could be attributed to the overapproximation or, alternately, this behavior could be explicitly mentioned as a known side-effect behavior.

7. Implementation

Because the implementation of solutions presented in Chapter 6 consists of thousands of lines of code which rely heavily on understanding the Python scanner in a great detail, we are only going to point out and explain the most important implementation details in this chapter.

In most cases, however, once the solution is designed, the implementation does not pose any challenge and it is the amount of code needed to be altered rather than the algorithmic complexity that makes this task difficult.

7.1 Column handling

The implementation of the column handling feature was the most demanding since the Python scanner already contained an implementation of database column handling. Therefore, the implementation did not only require extending the scanner, but it also required some changes in already existing code.

7.1.1 Column handling classes

To cover many use cases of the column handling, several classes had to be implemented. In this section, we are going to describe them and provide examples of code when they are used. To maintain readability of classes, only key functionality and attributes of classes and interfaces are going to be present in diagrams for the rest of this section.

PythonColumnParent

We are going to start with the first flow type in the data flow life cycle - the `PythonColumnParent` interface. As you can see in Figure 7.1, there are three extending classes for database, file and console inputs. They map to the three equally-named standard resources used in other MANTA scanners.

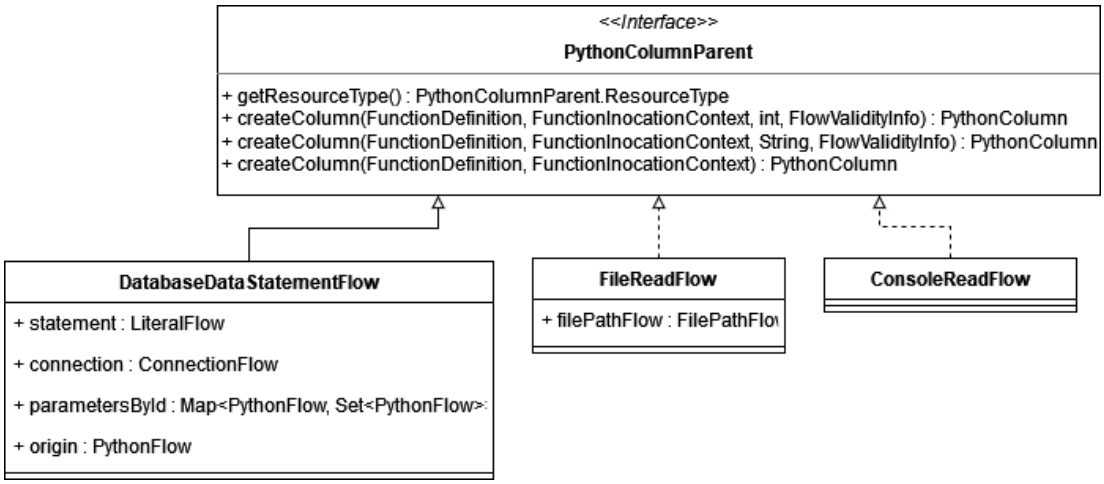

Figure 7.1: Classes implementing the `PythonColumnParent` interface and their most important attributes.

DatabaseDataStatementFlow Represents the database that is being used for loading data via an SQL query. It carries information about the database connection. In addition to that, it is also capable of storing the SQL statement being executed and the parameters being used (if there are any). However, the functionality tracking SQL parameters is not currently fully supported.

As you can see in Figure 7.1, the class has got an origin attribute, which makes no sense for a data source. We will clarify this field later in this section.

FileReadFlow Keeps information about the data source which is, in this case, a file, identified by its file path. Because the file path does not only have to be in the local file system, resources which can be defined by a path or an URI can be represented by this flow as well, for example, an Amazon S3 resource.

ConsoleReadFlow A data source coming from the console. This input has never got any indexed or named columns and, therefore, all three variants of the `createColumn` method return the `ConsoleReadFlow` instance itself. We explained why console resource and its columns can be represented by the same object in Section 5.1.

PythonColumn

As we outlined in Section 6.1.7, there are more column types than there are column parents or resource terminals. The reason for this is simple - while every resource has to have a column that represents the resource data, columns do not necessarily have to have a resource they belong to.

As an example, we can use library-specific columns which would typically have resource columns as their source of data (e.g. file or database columns), but the library itself is never going to be a resource - we can not write data into a library or read from it.

Therefore, there are two general types of columns - resource and library columns, you can see them in Figure 7.2. First, we will describe the widely-used resource columns, and then we will have a look at library columns.

Because column resource is identified by the `PythonColumnParent` object that is their (transitive) origin, we do not have to implement indexed, named, or unknown columns for file and database separately, but we can unify them and reuse these objects for both situations.

IndexedResourceColumnFlow Represents a resource column which is identified by an index (a numeric value) within its resource (a file or a database).

NamedResourceColumnFlow Represents a resource column which is identified by a name (a string value) within its resource.

UnknownResourceColumnFlow Represents a resource column which has got an unknown identification within its resource.

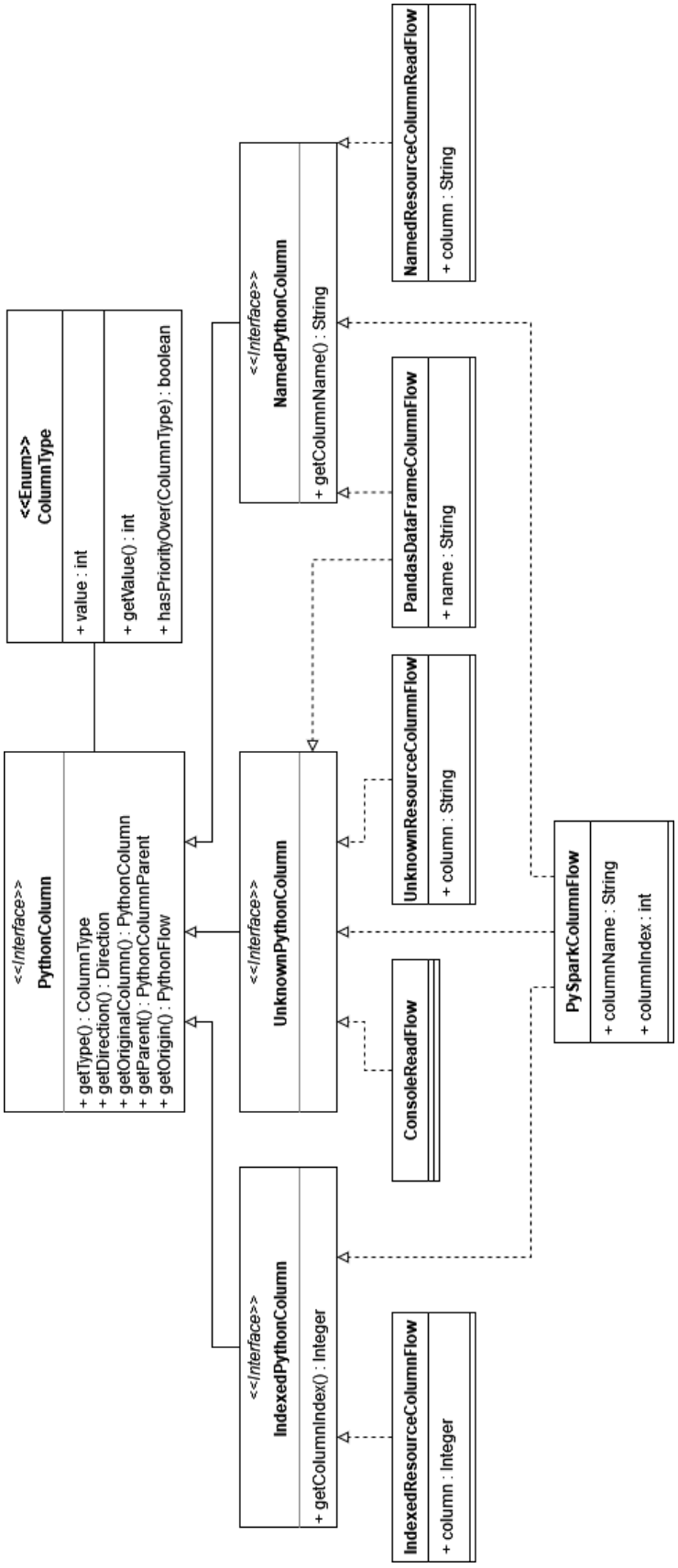

Figure 7.2: Classes implementing the `PythonColumn` interface and their most important attributes.

ConsoleReadFlow As was the case for the `PythonColumnParent` interface, this flow also implements the `PythonColumn` interface, which helps with simplification of the implementation of both propagation modes and transformation of flows into the data lineage graph.

PandasDataframeColumnFlow The first plugin column is the one for the usage in the analysis of the pandas library. Library columns can implement more than one `PythonColumn` interface's sub-type because this makes the analysis easier and there can usually be more than a single identification for some columns.

In this case, the `PandasDataframeColumnFlow` class implements both `NamedPythonColumn` and `UnknownPythonColumn` interface. If the pandas column's name is not known, it is treated as an empty string. The type of the column is also determined according to the non-empty property of the column name.

Similarly to the PySpark column, this column is also represented as-is without the wrapping DataFrame flow. We explained this design in Section 6.3.1.

PySparkColumnFlow Finally, the newly introduced column for the PySpark plugin. It implements all three column sub-type interfaces. It can be identified by its position within the DataFrame, its name, both, or none of them. These four states are determined by checking whether the index and name are equal to their default values (-1 for index and an empty string for the name).

We will discuss the `PySparkColumnFlow` class and its usage in the plugin later, in Section 7.3.

PythonResourceTerminal

At the end of the data flow life cycle, we have got four terminal flows, as you can see in Figure 7.3. When compared to `PythonColumnParent` interface, there is one extra flow, which covers a special case that appears, so far, only in the PySpark plugin.

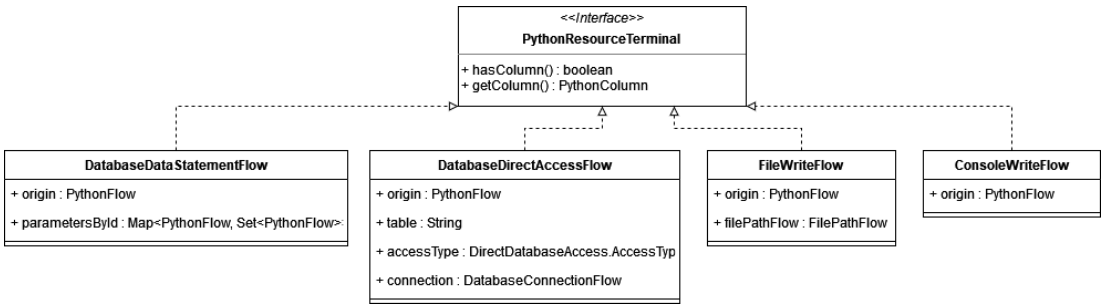

Figure 7.3: Classes implementing the `PythonResourceTerminal` interface and their most important attributes.

DatabaseDataStatementFlow We already described this class among column parent classes. For this case (output), there is the origin attribute present. This attribute stores the flow object being written. The rest of the class and its usage is equivalent to the usage as a column parent.

DatabaseDirectAccessFlow A special use case for database access. It does not contain any SQL statement, instead, it specifies that data is written to a specific column in the table of the defined database connection. Direct database access is to prevent generating non-existent SQL statements upon column writes - there are no SQL statements present in the code for DataFrame writing, however, we are capable of specifying (or *estimating*) which columns are being written.

FileWriteFlow Similar to the `FileReadFlow` class with one difference - this class allows for setting the origin, which is the column being written to the file represented by an instance of this class.

ConsoleWriteFlow Represents a write to the console. This class can *write* any column (or flow) and, as opposed to the `ConsoleReadFlow` class, it does not only have to be an unknown column, although the console output in the graph would only contain a single column.

7.1.2 Transforming columns into data lineage graph

Due to the design of columns, it is relatively simple to transform the flows resulting from dataflow analysis into the graph. We need to start the transformation from the end and construct the data lineage backwards.

Therefore, for every `PythonResourceTerminal` instance, we:

1. Create an output node for the resource.
2. Create the written column node within the resource-representing node.
3. Find the *original column* of the written column (see Section 6.1 for more details on how original column is found).
4. If not found, do not create any input flows and finish.
5. Otherwise, get the original column's parent, an instance of the `PythonColumnParent` interface, and create its graph node.
6. Create the read column node within the original column's parent node.
7. Add an edge between the read and written column nodes.

For different column, column parent, and resource terminal class types, there are different handlers for creating edges, for example, because a file-representing graph node contains different data than the node representing a database, but the implementation has only got small differences between handlers and the code is generally shared among individual handler classes.

No further changes were necessary except for the implementation of direct database access in the common Dataflow Generator for all intermediate language scanners (C#, Java and Python).

7.1.3 CSV Column Recognition

The implementation of the CSV column recognition in the Python scanner required no design as this feature used the designed column handling in a straightforward manner. We will describe in this section what happens in the scanner when a reader or a writer is used in the analyzed source code.

Using CSV readers

When a new instance of a reader is created (for both common and dictionary readers), a new `FileReadFlow` instance is created, using file path(s) present in the constructor parameter. Then, an unknown column is created for every read flow - we have no information about the structure of the read file. This file is then propagated to the return value of the constructor, assigned to the `reader` variable in the code sample below:

```
# open() creates a file path and assigns it to csvfile
with open('input.csv', newline='') as csvfile:
    # common reader
    reader = csv.reader(csv_file, delimiter=",")
    # dictionary reader
    reader = csv.DictReader(csvfile)
```

The reader instance's unknown column can then be used in the rest of the code as the representative of the data read from the CSV file. Rows can, then, get more granular by invoking the `__getitem__()` function over the reader output. In such situation, we can create a named, or indexed column, with the origin being the unknown column from the reader.

```
# common reader
# returns a list with splitting row by comma, indexed by a number
for row in reader:
    print(row[0])

# dictionary reader
# returns a list with splitting row by comma, indexed by a string
for row in reader:
    print(row['first_row'])
```

From the code above we can see deduce that in the first case, we only write the first column, while in the second case, it can be identified that there is a column named *first_row* and it is being printed into the console. We will show how these examples are visualized in MANTA graphs in Chapter 8.

Using CSV writers

The scenario with writers is very similar to that of readers. As we described in Section 5.1.2, it is only possible to write with CSV writers using functions `writerow` and `writerows`. `Writerows` is, essentially only bulk `writerow` and, therefore, a lot of analysis can be performed the same way.

When writers are created, the scanner only stores the path of the output file to the writer-containing variable:

```
# open() creates a file path and assigns it to csvfile
with open("out.csv", "w", newline="") as file:
    # only propagate path from 'file' to 'write', no modifications
    writer = csv.writer(file, delimiter=",")
    writer = csv.DictWriter(file)
```

This is because we only create the resource terminal (in this case, of the `FileWriteFlow` class) after we get the column being written (remember the data life cycle from Figure 6.3). Therefore, we must keep the path and wait for the write operation's invocation which will contain the data being written into the file.

Once the function `writerow` is invoked, the propagation mode we implemented is used and the input data is analyzed. The function only allows for an iterable object to be written, such as a list, a tuple, or a dictionary. Therefore, we can determine the position or the name of the column to which each item is being written from the key of its `DictionaryItemFlow`[1] object. Then, only these dictionary items need to be processed and a quite accurate data lineage can be created. With the index or the name known, a new indexed/named resource column is created and this new column is, then, the *origin* of the newly created `FileWriteFlow` instance, which can now be created, when the written column and file path are both known.

For the case of `writerows`, it is necessary to get the value of every dictionary item twice, because this function's parameter is expected to be an iterable of an iterable, for example:

```
data = [
    ['Alabama', 28748, 'AL', 'ALB'],
    ['Algeria', 2381741, 'AG', 'DZA']
]
```

Therefore, we have to treat every iterable in the `data` variable as a separate invocation of the `writerow`. This is the part where the functionality can be shared.

One last corner case is that a string is also an iterable object. To cover this case, the scanner simply creates an unknown column, instead of an indexed or a named column. There is no useful information which can help with the identification of the column. Below, you can see a situation in which this scenario can happen:

```
# written data is from the console input
data = input('Enter data to be written')

# create a file path flow
with open("out.csv", "w", newline="") as file:
```

[1] A `DictioanryItemFlow` represents an element of a collection with two values - its key (index in a list, a string or any other value in a dictionary), and its value. All dictionary items form a dictionary by simply being associated to the same expression. In code `li = ['a', 'b']`, variable `li` would contain two dictionary items - one with key 0 and value 'a', the other with key 1 and value 'b'.

```
# assign path to 'writer'
writer = csv.writer(file, delimiter=",")

# create an unknown column with origin being console input
# and assign this column to a file write flow
writer.writerow(data)
```

7.2 Flow Variables

The implementation, based on the design, is very simple. All modifications that have been made in the source code match the changes proposed in the design of this feature in Section 6.2 and, therefore, it makes no sense to repeat it. Instead, we can demonstrate how it is utilized in the PySpark structures designed in Section 6.3. In Section 6.3.3, we discussed that flow variables were going to be important in order to resolve the SparkSession's `read` and DataFrame's `write` attributes.

The flow variables feature was developed in order to be easy to use and, as we will show in code snippets of this section, it only requires a couple of lines of code to configure.

Let's start by having a look at the commands necessary for obtaining a `PySparkReaderFlow` instance from the `PySparkSessionFlow` object via an access to its `read` property.

The flow variable setting is, therefore, very simple. When the `read` attribute is accessed, the session instance creates a new reader instance, exactly how it is done in PySpark upon every property access and, this new reader flow, is returned. When the `PySparkSessionFlow` object is created, the constructor only needs to register a single entry into its flow variables in order to make the field access work properly. The needed command can be seen below.

```
addInnerFlowMapping("read", key -> Collections.singleton(
        new PySparkReaderFlow(AExecutableFlow.START_NODE_INDEX,
            executable, this)));
```

As you can see, even though the function accepts a single argument, it is not used as there is no need for that. The `PySparkReaderFlow` class requires a session argument in its constructor, which, currently, does not have much use. However, it is implemented in this way due to the fact that it is likely that sessions are going to be tracked in the future to provide a more detailed analysis. We discussed sessions in Section 3.4.5.

A similar approach is used for the `write` property access of DataFrames. For every column present in the DataFrame that's writer is needed, the column shall return itself as a part of the writer's referenced DataFrame. Therefore, every `PySparkColumnFlow` object simply needs to return itself when it is queried for the `writer` property:

```
addInnerFlowMapping("write", key -> Set.of(this));
```

This way, all columns are successfully propagated to writing functions of the `DataFrameWriter` class. If it was not for this flow variable configuration, no columns would be available and, therefore, data lineage information for the write operation would be lost.

There is one more use case for the flow variables in PySpark's columns. Since columns can be obtained by field access: `df.column` where *column* is a name of a column in the target DataFrame, a simple configuration of flow variables enables this functionality also in the Python scanner's analysis.

All that is needed is to add a new mapping, where the key of the mapping is the name of the column and the returned value is the column flow instance itself, as you can see in the code snippet below. Note that this only works for named columns and for other column types, this configuration is omitted.

```
if (!getColumnName().equals("")) {
    addInnerFlowMapping(getColumnName(), key -> Set.of(this));
}
```

When the column is, then, queried, the analysis iterates over all flows of the DataFrame, in our case over all column flow instances, and tries to find the one with the correct name, using flow variables. All columns return an empty set except for those that are wanted. Thanks to flow variables and a single command, it was possible to support another commonly used feature of PySpark's DataFrames.

There are many more use cases for this feature in the future, however, for now, only the PySpark plugin makes use of it.

7.3 PySpark Plugin

The implementation of the PySpark plugin contains the four core model classes designed in Section 6.3: `PySparkColumnFlow`, `PySparkReaderFlow`, `PySparkSessionFlow`, and `OptionFlow`. These structures are implemented according to the design and, therefore, there is nothing relevant to point out except for the flow variables usage, which was already described in the previous section.

Instead, in the following sections, we will focus on several implementation specifics not mentioned before.

7.3.1 Column builder

Because of the way how columns are chained to each other during analysis to track changes of columns, it happened many times that PySpark columns had to be duplicated with very small changes, for example, only changing the name of the column or its index. Since this task was very repetitive, creating long constructors due to the fact that PySpark columns contain a lot of information, an auxiliary column builder was created, named the `PySparkColumnBuilder`, which simplifies this duplication.

In principle, it can be seen as a modifiable, temporary, version of the `PySparkColumnFlow` class, which is used during propagations, but never propagated out of a propagation mode - that is also why it is not a *flow*, to prevent it from being used incorrectly.

The builder contains a couple of factory methods for its initialization - either to initialize from an existing PySpark column or to initialize from scratch:

```
public static PySparkColumnBuilder from(
            PySparkColumnFlow originalFlow)

public static PySparkColumnBuilder from(
            FunctionDefinition functionDefinition)
```

These are the only two ways how to obtain an instance as the constructor is private. Once an instance is obtained, it can be modified using simple setter-type methods utilizing the fluent interface by returning the instance itself. There are methods to change the name, the index, or, for example, to add a column alias:

```
public PySparkColumnBuilder withName(@NonNull String name)

public PySparkColumnBuilder withIndex(int index)

public PySparkColumnBuilder withColumnAliases(
            @NonNull Collection<String> aliases)
```

When all modifications are done, the builder is used to turn it into a `PySparkColumnFlow` instance using the `build()` method. There are different version of this overloaded method for different use cases. The non-parameterized one can be used when the origin of the new column shall be the original one (or no origin at all in case the builder was not initialized from an existing instance of a PySpark column flow). Other versions allow for defining the origin(s) of the newly create column(s).

The builder is easily extensible, so only a new field and a setter method are necessary if the `PySparkColumnFlow` class adds more properties in the future. Thanks to this solution, the implementation of propagation modes has gotten simpler and easier to read.

7.3.2 Reading and writing of DataFrames

When we look at how DataFrame read and write operations work in general, they are very similar. First, there needs to be some analysis of options present in the reader or writer - the target format, source/target file path or database details. Then, the format is used to distinguish between file- and database-based operations. For each of these operations a different action is performed as they work with different data strucutres.

However, the part of option resolving and the overall division among file- and database-based operations is common regardless of the fact whether we are analyzing a reading or a writing operation. Therefore, it makes sense to create a propagation mode that would take care of these common parts once, reducing code duplicity. For this use case, the `PySparkIoPropagationMode` class was created, which provides its extending classes a method named `createResourceFlows()` performing three steps:

1. Collect options of the I/O operation

2. Categorize the format and decide between a file- and a database-based operation
3. Create resources per selected operation
4. Return created resource flows

The third step, creation of resources, is entirely configurable by declaring both file- and database-based operation as abstract methods, enforcing an implementation by extending classes. This makes the `createResourceFlows()` a template method.

It means that for reading operations, the propagation mode handling this part of analysis must define how file read flows are created and which options are used for this, and similar stands for database resources - it defines how database statement flows are initialized as well. Everything else, though, is handled by the common propagation mode.

For write operations, the situation is similar, as the `PythonResourceTerminal` flow creation needs to be defined for both resource types (console is, obviously, ignored). However, write operations do not propagate any flows since they are effectively terminal in the life cycle of columns within the Python scanner analysis. Therefore, the output of step four of the template method is ignored.

Due to this implementation, the read- and write-operation-handling propagation modes can focus on handling their specific tasks and the common part is handled separately in their common abstract base class.

7.3.3 PySpark transformations

At the beginning of Chapter 5, we pointed out a few DataFrame transformations which we deemed the most important in terms of data lineage analysis of the new PySpark plugin for the Python scanner. Analysis of some of these transformations was simpler, but in some cases, several different scenarios had to be taken into account. Below, we describe every transformation propagation mode and its steps. We briefly described these functions in Section 3.4.2.

crossJoin()

Same as `join()`.

drop()

No action for this transformation. The problem is that for deleting operations, like this one, there is no data flow present. In order to keep the overapproximation approach to the analysis, no changes are made upon invocation of this method in the source code. A problem may occur if the parameter of the target drop column contained more than one value and, as a result, more columns would be dropped than intended and, as a result, the data lineage would lose some data flows. Consider following source code:

```
df = ...              # contains columns col1 and col2
```

```
my_var = 'col1'
... some operations ...
my_var = 'col2'      # my_var contains flows of both col1 and col2

df2 = df.drop(my_var) # col1 is incorrectly dropped
```

In the source code above, due to the fact that flow sets of variables can only be extended, not overridden or have their flows dropped, the `my_var` variable would contain constant flows for both `col1` and `col2` before the `drop()` function is analyzed. As a result, if we processed the drop transformation, two columns would be matched and dropped. even though only a single column is dropped in the source code.

This is an incorrect propagation and in case the resulting DataFrame was written into a data source, the `col1` column would be missing in the data lineage graph. Therefore, there is no propagation for this function.

join()

The join statement aggregates columns of two DataFrames using a join expression. While there is no change needed for the receiving DataFrame no matter what, the joined DataFrame's columns are appended and the columns from joined expression are merged.

Since the evaluation of expressions is not supported by the Python scanner, we can only claim that all columns are simply appended, including columns on which the join is performed. Because we may not have a complete information about the amount of columns in the receiving DataFrame (for example, when we only deduce some columns without a provided schema), the precise position (index) of joined columns can not be set. Therefore, appended columns have got their indexes removed.

select()

In case of the `select()` function, two types of parameters can be passed - either PySpark column instances or just their names. Because the function uses the arbitrary parameter `*cols`, the Python scanner wraps all parameters into a collection and when the propagation starts, it first must unpack all collection items. Then, matching is performed:

1. *Matching by column instance* - the passed columns are matched against columns present in the DataFrame and if they match the selection, they are returned.
2. *Matching by column name* - this matching has got two different scenarios:
   (a) An asterisk is encountered - simply return the selected DataFrame as-is.
   (b) An asterisk is not encountered - three types of columns are returned:
   - Columns whose name matches at least one of the columns to be selected.

- Columns whose at least one column alias matches at least one of the columns to be selected.
- Unknown columns - to cover these columns within overapproximation, mapped to columns with the searches names.

Columns returned from both matching approaches are, then, returned. Since selecting is highly schema-oriented, this propagation may cause significant overapproximation. However, it is necessary in order to capture all possible scenarios.

union()

In PySpark, the `union()` transformation is based on the merged columns' index, instead of names. This means that columns need to be matched with their columns and columns with an unknown index need to be considered as well. The propagation mode works in several steps:

1. Collect all columns of the receiving DataFrame - columns into which is the other DataFrame unioned - later named *receiver columns*.
2. Collect all column of the *other* DataFrame - the one being unioned - later named *other columns*.
3. Categorize all origins of *other columns* into sets per index (position) - columns without an index are all at index -1 as *unknown position*. These origins shall be added into every column (overapproximation).
4. Add all column sources from *other columns* matching the given index to all *receiver columns* with matching indexes, creating a their copy with assigned proper origin of data. Same for columns with an unknown index.
5. If there is an unknown column among *receiver columns*, add all *other columns* to that column as its origin via making a copy and assigning it as the copied *receiver column's* origin.
6. Add the newly discovered columns from *other columns*, not present in the original flow. If there was and unnamed-and-unindexed column among *receiver columns* before, it was already merged with matching *other columns*. If there was no such column it is created in this step as an overapproximation.

Due to a high chance of missing parts of DataFrame schema data, this propagation is very complex and needs to cover many combinations. Again, in some situations, this may cause a considerable overapproximation.

unionAll()

Same as `union()`.

withColumn()

A simple propagation with two arguments - the name of the added column and the instance of the column to be added. It is necessary to collect the added column's name(s) and the column(s) to be added. Then, permutations are made for every added column and every column's name, when the column is simply renamed and added to the DataFrame.

withColumnRenamed()

Again, a simple propagation. Collect the name(s) of the columns to be renamed. Then, collect the name(s) to which the matching columns shall be renamed. Then, only an iteration of the DataFrame's columns is necessary and a change of name is needed for all columns matching the name(s) of the renamed columns. For all matching columns, new columns have to be created for every new name found. Because of the same scenario as for the `drop()` function, no columns are removed, only the renamed columns are added.

7.3.4 Testing

As it is common in the Python scanner, there are three levels of testing related to the new PySpark plugin:

1. **Unit tests** - test that propagation modes implemented in the plugin work as expected for various inputs, corner cases and correct deduction when certain pieces of information are missing is tested.

2. **Integration tests** - realized on the level of the Python reader. The whole process from parsing, through alias and symbolic analysis, until the result transformation is executed. These tests make sure that all scanner's components used by the reader in data lineage analysis work properly and the result matches the expectations.

3. **System tests** - implemented in the module which configures the common *Dataflow Generator* to process the outputs of the Python scanner's reader. We mentioned the common Dataflow Generator in Section 2.2.3. These tests verify that all outputs produced by the analysis of the Python scanner are transformed correctly by the common generator component and the actual graph, which is visualized in MANTA Viewer, looks as-expected.

   These tests utilize other MANTA scanners and components, for example those, which handle creating database or file system nodes and edges.

There are around two hundred unit tests and in case of integration and system tests, only a few testing scenarios verifying the functionality of the PySpark plugin are present. Both integration and system test inputs are Python scripts and, therefore, it is possible to test the whole plugin in only a few scripts which use several functions handled by the PySpark plugin and ensure that inputs and outputs of propagation modes work well together and no data lineage is lost.

According to SonarQube, the code quality analysis tool used in MANTA's CI/CD pipeline, the code coverage of unit tests is 43,5% for the PySpark extractor, and 96,4% for the PySpark dataflow plugin. The low extractor coverage

is mostly caused by the fact that extractor tests are, in general, executed in the Python scanner's extractor-core module, and this coverage is not taken into account for the specific extractor module.

8. Evaluation

In this chapter, we would like to demonstrate that the features implemented in this work, namely the CSV column recognition, flow variables, and the PySpark plugin, work and produce the expected data lineage graph.

We are going to use two separate examples as there is not much connection between the source code using CSV reader and/or writer, and the PySpark functionality. First, we are going to present, explain, analyze, and show the result of the CSV column recognition functionality, and then, we will repeat the process with the PySpark input source code.

Note that the analyzed source code is created in order to highlight the new scanner functionality and it may not entirely make sense to execute it in a production environment. However, with minor adjustments, both examples could make sense even to be used by users in real-world use cases. Presented scripts can be executed without an error given the file and database environment needed is correctly prepared. In terms of the Python syntax, all source code is valid.

8.1 CSV column recognition example

As mentioned earlier, we are going to focus on the demonstration of the CSV column recognition first. As you can see in Figure 8.1, the code is not very long and is limited to only focus on the functionality we implemented within this work, which allows us to see clearly how the scanner behaves in reality.

First, on line 4, the example program opens a file named *names.csv*, which is the input data source for the instantiated dictionary CSV reader on line 5. Then, on line 7, another file *output.csv* is opened, which is used as the target file for the common CSV writer on line 8. With the reader and the writer created, the actual data operation is executed on lines 10-12, where the reader is used to read all input data source lines iteratively, with each line's values of columns *last_name* and *first_name* are written into the output file, using the `writerow()` function.

In case of the reader, its string constant access option is used, while for the common writer, values are passed by position in a tuple. We explained how CSV readers and writers work in Sections 5.1.1 and 5.1.2.

Scripts like this can be used in situations when data sorting, filtering, or cleansing is desired in order to create data sets for certain data marts. If a system outputs a lot of data into CSV files, but only certain columns are needed for a particular data set, CSV reader and writer may be used.

Having the input source code explained, let's have a look at how its data lineage graph.

You can see the output of the output graph of the analysis of the Python scanner in Figure 8.2. On the left- and right-hand size, nodes in the shades of the green color are file system nodes, while yellow- and black-colored nodes represent the Python technology in this graph visualized by MANTA Viewer.

On the left side, we can see that two columns are being read, which is the information deduced from the invocation of the `__getitem__()` function on lines 11 and 12 (string constants in brackets). These two columns are loaded into the Python column, note that the scanner determines the line of this read operation

```
1  import csv
2
3
4  with open('names.csv', newline='') as infile:
5      reader = csv.DictReader(infile)
6
7      with open('output.csv', 'w') as outfile:
8          writer = csv.writer(outfile, delimiter=',')
9
10         for row in reader:
11             writer.writerow((row['last_name'],
12                  row['first_name']))
```

Figure 8.1: Source code analyzed by the Python scanner to demonstrate the CSV column recognition feature.

to be four, instead of eleven. That is because the file operation, which wraps the columns, began on line 4, when function `open()` was invoked.

Edges between the two groups Python nodes in the middle of the figure show that the values in the first column originate in the input file's column *last_name* and values in the second column originate in the input file's column *first_name*, just as we would expect. Note that indexed columns start with number one, as it is defined in MANTA that the unknown column in nodes of the Python technology is marked as zero, while in file system nodes, the unknown column is marked as number one.

This is also the reason why columns one and two from the Python output lead to columns two and three in the output file. When the user views the graph, they can see that if a system (BI, or ETL tool, other program) loads the data from the file *output.csv*, it actually reads the data that originate in *input.csv*. Column deduction yielded expected results and the data lineage graph provides as detailed information as it is possible to retrieve from the source code.

8.2 PySpark plugin example

Now, let's get to the second example, which focuses on demonstrating how the PySpark plugin works. The source code for this example is in Figure 8.3. It is much longer as there is a lot of functionality to demonstrate, although not all implemented propagation modes are used in order to maintain readability of the input and also the output data lineage graph. Equally to the previous example, this program can also be seen as a script to perform a simple ETL[1] task within a larger ETL pipeline.

Before we present the result of the data lineage analysis, let's analyze the input. The script can be split into five main parts:

[1]Extract, transform and load. It is a common data process which names tasks which, in general, are performed three stages. The first stage loads a data from a source, the second stage performs a certain data operation, and the last stage writes the output of the operation into another data source

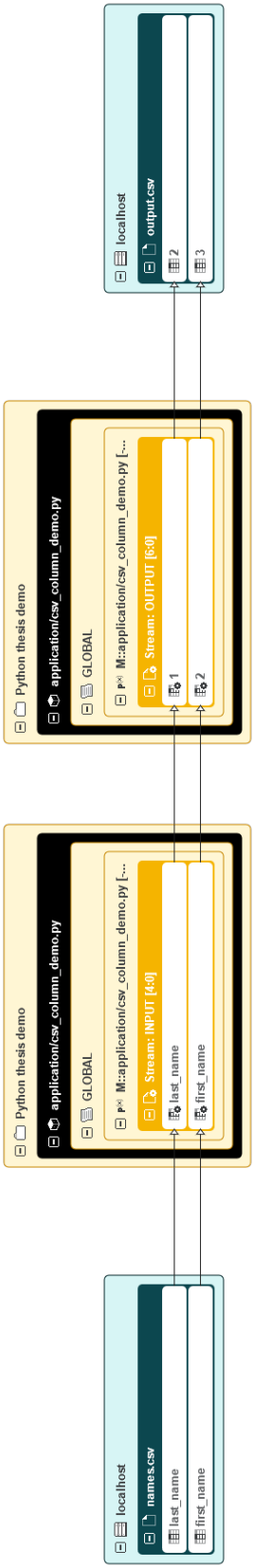

Figure 8.2: Data lineage graph of the input from Figure 8.1.

1. Lines 1-4: **PySpark imports** - this is the least important part, importing the essential functionality from the PySpark SQL module.

2. Lines 6-11: **Defining DataFrame schemas** - to show how column deduction is also possible in the PySpark plugin, two DataFrame schemas are defined first which will be used later in the code. The first schema, `db_schema`, contains two fields - *FULL_NAME* and *COUNTRY*. The second schema, `csv_schema`, contains columns *COUNTRY* and *CUST_CLASSIFICATION*. There is an equally-named column present in both schemas which will be utilized later.

3. Lines 13-24: **DataFrame initialization** - in this part, two DataFrame instances are created. First, the `db_df` is created by loading a database table from a resource defined by a JDBC connection string and using the `db_schema` defined in the previous step. The second DataFrame, `csv_df` is created by using the CSV file data source *./countries.csv*, using the `csv_schema` schema.

4. Lines 26-28: **DataFrame operations** - here, the DataFrame instances are modified, a simple data transformation is performed. First, a new column named *REGION* is added into the `csv_df` DataFrame. Then, the column *FULL_NAME* of the `db_df` DataFrame is reanamed to *CUSTOMER_NAME*. Lastly, a new DataFrame, `joined_df`, is created by joining `csv_df` to `db_df`. Remember that the join operation is matched by column name, as we described in Section 7.3.3, so the *COUNTRY* column of both schemas shall be merged into a single one.

5. Lines 31-38: **DataFrame writing** - the last part of the example source code performs two write operations of the created `joined_df` DataFrame. First, there is a write operation of all columns into a CSV file *joined_df_file.csv*, and then, columns *CUSTOMER_NAME*, *COUNTRY*, and are selected and written into a database table *dwh.CUSTOMER_CAT*.

Now that we have explained what the example source code does, we can show and describe the output data lineage graph, which is shown in Figure 8.4. The color scheme of nodes per technology remains the same, with the light-green color representing the MS SQL technology.

As you can see in the graph, loading of the `db_df` DataFrame is shown in the top-left part of the graph (bottom-left corner of the page). The PySpark plugin correctly deduced that two columns *FULL_NAME* and *COUNTRY* were loaded from the specified database data source and the table. However, due to the overapproximation caused by the way how the scanner processes collections, two other columns are shown in the graph. On line 10 in the script, the two fields are passed to the `StructType` class constructor in a `list`, with indexes 0 and 1. However, the scanner passes this information after extracting the two `StructField` objects, which makes the plugin create two indexed PySpark columns and there is not much that can be done within the PySpark plugin to limit this behavior.

The initialization of the `csv_df` is shown in the bottom-right corner of the page, with, again, two correct columns, and two indexed columns resulting from the way that collections are processed.

```
1  from pyspark.sql.session import SparkSession
2  from pyspark.sql.types import StructType, StructField, \
3                                IntegerType, StringType
4  from pyspark.sql.functions import lit
5
6  field1 = StructField("FULL_NAME", IntegerType(), False)
7  field2 = StructField("COUNTRY", StringType(), False)
8  field3 = StructField("CUST_CLASSIFICATION", StringType(),False)
9
10 db_schema = StructType([field1, field2])
11 csv_schema = StructType([field2, field3])
12
13 spark = SparkSession.builder.getOrCreate()
14 db_df = spark.read \
15     .format("jdbc") \
16     .schema(db_schema) \
17     .option("url","jdbc:sqlserver://localhost;database=manta")\
18     .option("dbtable", "dwh.rep_client") \
20     .option("user", "username") \
21     .option("password", "password") \
22     .load()
23
24 csv_df = spark.read.csv("./countries.csv", schema=csv_schema)
25
26 csv_df = csv_df.withColumn('REGION', lit(None))
27 db_df = db_df.withColumnRenamed('FULL_NAME', 'CUSTOMER_NAME')
28 joined_df = db_df.join(csv_df)
29
30 # write resulting dataframes to CSV and database
31 joined_df.write.csv("joined_df_file.csv")
32
33 joined_df.select('CUSTOMER_NAME', 'COUNTRY',
34     'CUST_CLASSIFICATION').write \
35     .format("jdbc") \
36     .option("url","jdbc:sqlserver://localhost;database=manta")\
37     .option("dbtable", "dwh.CUSTOMER_CAT") \
38     .save()
```

Figure 8.3: Source code analyzed by the Python scanner to demonstrate the PySpark library support.

These two DataFrames are, then, mixed together in the fourth part of the source code. We can see that there is a newly created column *REGION* nested under the *Stream: Output [29:0]* node, which was created in the operation on Line 26. We can also see that there is a renamed *CUSTOMER_NAME* in both output Python nodes, which was renamed on line 27. Remember that we do not remove any columns even during the rename operation, as we described in Section 7.3.3, so the original column *FULL_NAME* is present as well.

The last important thing to notice in this graph is that there is a properly joined column *COUNTRY*, from line 28. This column should have its origin from both DataFrames, as they both contain a column with this name. Data lineage for this column is highlighted in Figure 8.5. As you can see, the *COUNTRY* column of both DataFrames is the origin of the written column with the same name from DataFrame `joined_df`.

Note that as part of approximation, all columns without a name of the joined DataFrame (the one passed as an argument of the function), are the source of this column as well - see columns *1* and *2* of the CSV-file-originating DataFrame.

As for the DataFrame written into the database, the plugin managed to correctly match selected columns to their instances in the written DataFrame and not include any other columns as part of an overapproximation.

8.3 Limitations and Future Work

As demonstrated in previous sections, scanner extensions designed and implemented in this work are capable of analyzing the source code containing constructs for which they were implemented on a satisfactory level.

8.3.1 PySpark SQL coverage

It is relatively common that there is some room for improvement and this outcome is no exception.

Because it turned out that the PySpark technology is too large to be supported entirely as a result of this work, the plugin is fairly limited in the amount of PySpark functionality it can process. This is, however, not a large blocker as extending the plugin is simple as core data structures are already implemented and only new propagations have to be configured or created.

8.3.2 PySpark tables

Despite the fact that the PySpark plugin supports a majority of reading and writing operations, there are still some that are left unhandled. It is especially the piece of functionality regarding PySpark's tables. While most data is persisted in a database or a file, it is possible to also persist data in tables managed by PySpark. This allows for an effortless usage of PySpark SQL queries as these saved table names can be used in queries as if they were created in a standard data resource.

DataFrames can also be created by specifying the table from which the new DataFrame shall be created, using the DataFrameReader's `table()` function. They can be written into a table via the function `saveAsTable()` of the class

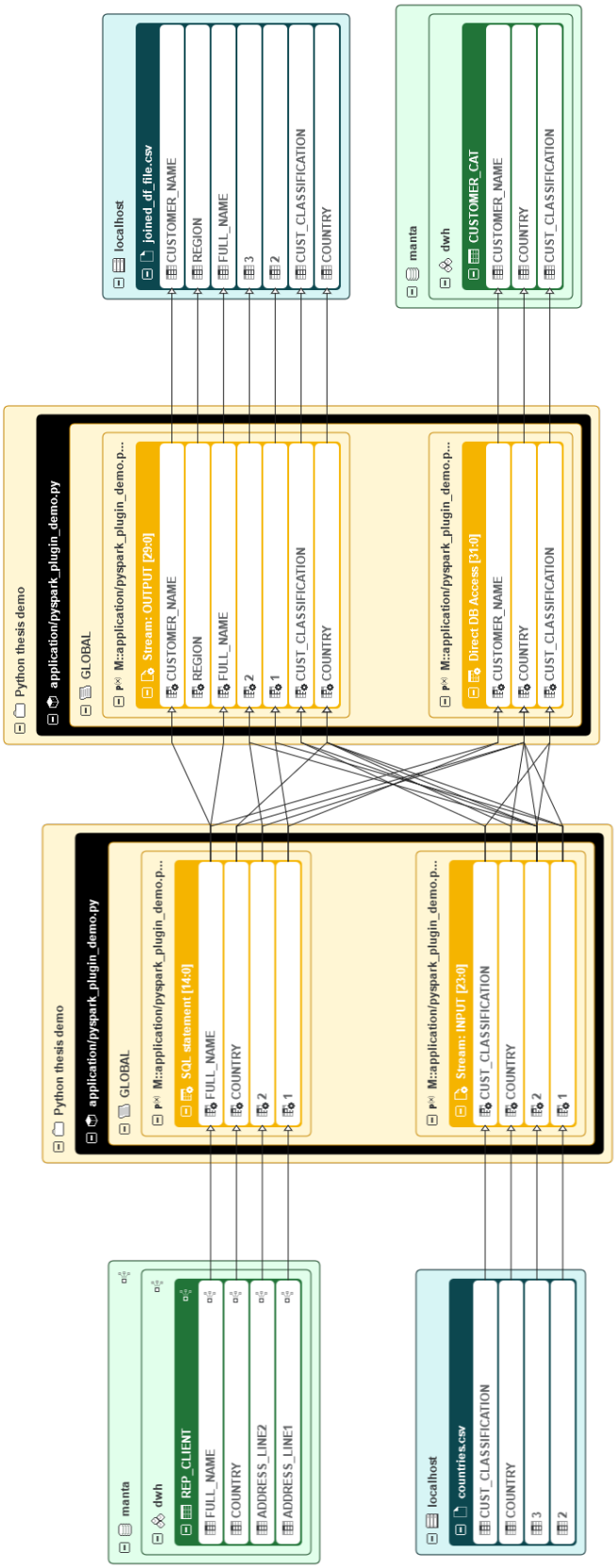

Figure 8.4: Data lineage graph of the input from Figure 8.3.

`DataFrameWriter`. While this is not a piece of functionality which we declared to support within this work, the plugin may miss some important data lineage information without the support for the analysis of these two functions.

8.3.3 Support for other PySpark modules

As it was mentioned several times in this work, the PySpark library comprises a lot of functionality split across several large modules and we only developed the plugin to support the elementary functionality of the PySpark SQL.

In the future, the plugin will have to be extended in order to be useful in more user scenarios and to be able to discover and analyze as much data lineage as possible, for example, to analyze data lineage when PySpark's MLlib module is used. This is, however, a subject to prioritization based on the requirements from users. The functionality needs to be chosen as it is almost impossible to cover the whole library.

8.3.4 Overapproximation

When we described data lineage graphs in the previous section, we often mentioned certain data flows which occurred as a result of an overapproximation. While it is impossible to entirely remove some incorrect data flows in data lineage graphs produced by the Python scanner, mainly due to certain operations depending on runtime information which is not available during the static analysis, it may be reduced.

In terms of the PySpark plugin, there are some pieces of propagation modes written a little too generally, for example, the processing of DataFrame schema, as we have shown in the graphs, or algorithms for matching DataFrames in join or union operations. The issue with the precision is also rooted in the fact that the actual behavior of PySpark in certain scenarios is not mentioned in the documentation and, therefore, a deep analysis of Spark code, which is called from within the PySpark library, is necessary.

This makes improving the precision of the scanner very difficult and makes room for imprecision. To mitigate incorrect behavior of the plugin, overapproximation is used to cover the actual data flows, even if that means that several incorrect data flows may be present.

8.3.5 No analysis of CSV reader and writer field names

One of the constructor parameters of the CSV dictionary reader and writer is named `fieldnames`. This parameter may be passed which defines the structure of columns being read or written. The scanner, currently, does not analyze this piece of information even if it is available, as the implementation of the support for CSV readers and writers was implemented to show that the column deduction works even in simpler cases than in the PySpark plugin.

Adding the analysis of field names to make column-level data lineage more accurate is not, however, very difficult and would not pose a major challenge to anyone who would try to add this feature into the Python scanner in the future.

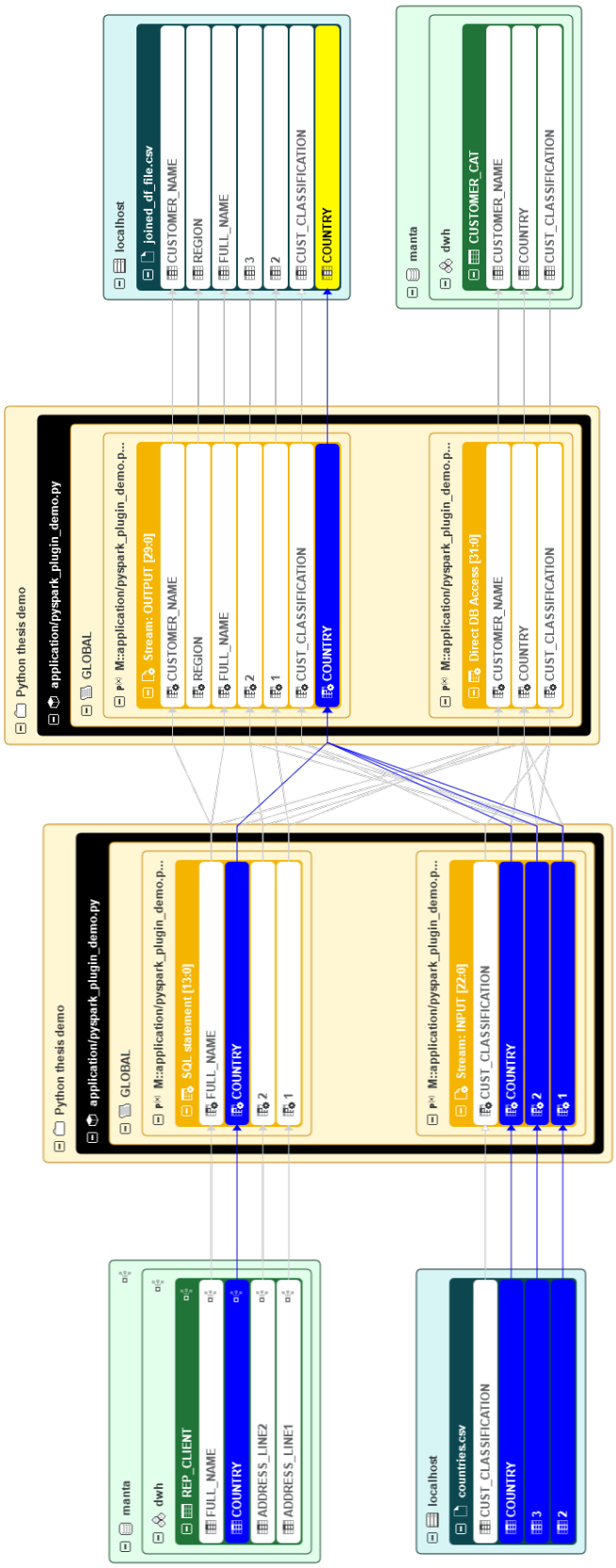

Figure 8.5: Data lineage graph of the input from Figure 8.3 with highlighted lineage for the column *COUNTRY*.

9. Conclusion

In this project, we have managed to successfully develop a PySpark plugin for the Python scanner capable of analyzing Python scripts which use the most common functionality of the PySpark library. This module is fully integrated in the scanner and it is a part of the MANTA Flow production deployment.

The plugin currently supports the elementary functionality of the PySpark library which on one side is able to detect and analyze data flows during its whole life cycle within the Python source code, but this analysis is limited by the number of supported input/output options and, additionally, by the amount of transformations which can be processed by the scanner. In the future, several extensions are going to be necessary as new use cases are going to be encountered in user source code. Since the library is enormous, it is almost guaranteed that new features are going to be required.

As we have shown in the last chapter, the plugin works as expected and is able to create the data lineage which employs all information available in the source code to produce as detailed graph as possible.

We also analyzed and designed a solution for the analysis of ORM source code. As it turned out, due to the heavy employment of type annotations in declarative ORM approach, it was not possible to implement support for the analysis of this technology in the SQLAlchemy library, which would require an additional implementation of processing of advanced Python features in the scanner's core and, therefore, extending the scope way past what was originally intended.

However, the design is detailed enough to not require any significant amount of time to design the last details before the implementation. As a consequence, when the blocking features are implemented, it shall be a fairy trivial task to implement the support as all major problems were already resolved in the design.

Bibliography

[1] *Basic Relationship Patterns - SQLAlchemy 2.0 Documentation*. URL: https://docs.sqlalchemy.org/en/20/orm/basic_relationships.html#one-to-many. (accessed: March 15, 2023).

[2] *Column Elements and Expressions - SQLAlchemy 2.0 Documentation*. URL: https://docs.sqlalchemy.org/en/20/core/sqlelement.html#sqlalchemy.sql.expression.text. (accessed: March 15, 2023).

[3] *Computing the Impossible: Static Analysis of Dynamically Typed Languages (Part 1)*. URL: https://medium.com/p/fb85a0fdd94. (accessed: October 29, 2022).

[4] *Data Lineage*. URL: https://www.techopedia.com/definition/28040/data-lineage. (accessed: October 28, 2022).

[5] *Data Manipulation with the ORM - SQLAlchemy 2.0 Documentation*. URL: https://docs.sqlalchemy.org/en/20/tutorial/orm_data_manipulation.html. (accessed: March 15, 2023).

[6] *Do you need an ORM?* URL: https://enterprisecraftsmanship.com/posts/do-you-need-an-orm/. (accessed: January 9, 2023).

[7] *Establishing Connectivity - the Engine - SQLAlchemy 2.0 Documentation*. URL: https://docs.sqlalchemy.org/en/20/tutorial/engine.html. (accessed: March 15, 2023).

[8] *Infographic: How much data is produced every day?* URL: https://cloudtweaks.com/2015/03/how-much-data-is-produced-every-day. (accessed: October 28, 2022).

[9] *Machine Learning Library (MLlib) Guide*. URL: https://spark.apache.org/docs/latest/ml-guide.html. (accessed: October 29, 2022).

[10] *Object-Relational Mapping (ORM)*. URL: https://www.techopedia.com/definition/24200/object-relational-mapping--orm. (accessed: January 9, 2023).

[11] *ORM Lazy Loading Pitfalls*. URL: http://gorodinski.com/blog/2012/06/16/orm-lazy-loading-pitfalls/. (accessed: January 9, 2023).

[12] *ORM Mapped Class Overview - SQLAlchemy 2.0 Documentation*. URL: https://docs.sqlalchemy.org/en/20/orm/mapping_styles.html#orm-mapping-styles. (accessed: March 15, 2023).

[13] *ORM-Enabled INSERT, UPDATE, and DELETE statements*. URL: https://docs.sqlalchemy.org/en/20/orm/queryguide/dml.html#orm-expression-update-delete. (accessed: March 15, 2023).

[14] *Overview - SQLAlchemy 2.0 Documentation*. URL: https://docs.sqlalchemy.org/en/20/intro.html. (accessed: March 15, 2023).

[15] *PEP 249*. URL: https://peps.python.org/pep-0249/. (accessed: March 15, 2023).

[16] *PEP 484 - Type Hints*. URL: https://peps.python.org/pep-0484/. (accessed: March 15, 2023).

[17] *PySpark Documentation*. URL: https://spark.apache.org/docs/latest/api/python/. (accessed: October 29, 2022).

[18] *Python Developers Survey 2021 Results*. URL: https://lp.jetbrains.com/python-developers-survey-2021. (accessed: October 29, 2022).

[19] *RDD Programming Guide*. URL: https://spark.apache.org/docs/latest/rdd-programming-guide.html. (accessed: October 29, 2022).

[20] *Reflecting Database Objects - SQLAlchemy 2.0 Documentation*. URL: https://docs.sqlalchemy.org/en/20/core/reflection.html. (accessed: March 15, 2023).

[21] *Spark SQL, DataFrames and Datasets Guide*. URL: https://spark.apache.org/docs/latest/sql-programming-guide.html. (accessed: October 29, 2022).

[22] *Spark Streaming Programming Guide*. URL: https://spark.apache.org/docs/latest/streaming-programming-guide.html. (accessed: October 29, 2022).

[23] *Spark Summarizer example - Spark ML*. URL: https://github.com/apache/spark/blob/master/examples/src/main/python/ml/summarizer_example.py. (accessed: January 9, 2022).

[24] *SQLAlchemy ORM - SQLAlchemy 2.0 Documentation*. URL: https://docs.sqlalchemy.org/en/20/intro.html. (accessed: April 17, 2023).

[25] *Top 10 Data Catalog Software Solutions*. URL: https://www.datamation.com/big-data/top-10-data-catalog-software-solutions. (accessed: October 28, 2022).

[26] *What is an ORM – The Meaning of Object Relational Mapping Database Tools*. URL: https://www.freecodecamp.org/news/what-is-an-orm-the-meaning-of-object-relational-mapping-database-tools/. (accessed: January 9, 2023).

[27] *Working with Database Metadata - SQLAlchemy 1.4 Documentation*. URL: https://docs.sqlalchemy.org/en/14/tutorial/metadata.html. (accessed: March 15, 2023).

[28] *Working with Database Metadata - SQLAlchemy 2.0 Documentation*. URL: https://docs.sqlalchemy.org/en/20/tutorial/metadata.html. (accessed: March 15, 2023).

[29] *Working with Transactions and the DBAPI - SQLAlchemy 2.0 Documentation*. URL: https://docs.sqlalchemy.org/en/20/tutorial/dbapi_transactions.html. (accessed: March 15, 2023).

List of Figures

A. Attachments

A.1 User Documentation

To run extraction and data flow analysis of the Python scanner module, several environment requirements need to be fulfilled:

- Java 11 needs to be installed on your computer.
- MANTA Flow has to be installed on your computer. This can be a major problem since only customers and developers of MANTA usually have access to this program.

A.1.1 Building the project

Our code consists of the PySpark plugin module integrated into the Python scanner and several modified files across the whole scanner which allow for a proper functioning of the plugin and other features implemented in this work. Since the scanner is already part of an experimental release of MANTA Flow, no building of the project is necessary as it is already present in every installation of the software. During the deployment of the Python scanner into MANTA Flow, all modules of the Python scanner are built to ensure that all module dependencies are satisfied.

A.1.2 Running the Scanner Module

If a user has got the MANTA Flow installed on their computer, they need to create a new Python connection, which defines whence the analyzed source code shall be extracted. After the extraction is finished, the user may edit the extraction configuration to select the module, or the function, where the analysis shall start. After that, extraction must be run to save changes and then, the DataFlow analysis scenario may be run, which launches the analysis of the extracted Python source code.

The output of the analysis can then be visualized and examined by the MANTA Flow Viewer, as we have shown in some of the figures used in this thesis work.

A.2 Contents of the Attachment

The attachment comprises following items:

The *source-code/Connector* folder, which contains the source code produced within this work. Note that these are just classes and other pieces of code written entirely within this work, not the whole Python scanner and, therefore, a lot of related source code is missing. The source code is hierarchically split into modules and components as they appear in the production code.

Additionally, files which were modified, but not written entirely, are not included. This is, mostly, related to two features - *Flow variables* and *CSV column recognition* as they mostly modified the existing implementation and classes.

Changes made to implemented these two features can be found in the following *diff* files:

- *source-code/csv-column-recognition-implementation.diff*

  Contains changes for the *CSV column recognition* feature implementation.

- *source-code/flow-variables-implementation.diff*

  Contains changes for the *Flow variables* feature implementation.

The *thesis.pdf*, which is the PDF version of this text.

The *tex-source* folder, which contains the TeX source code used for compiling this text. If it is needed to inspect some large diagrams or other figures in a greater details, all of them are present in this folder, in the sub-directory named *img*.