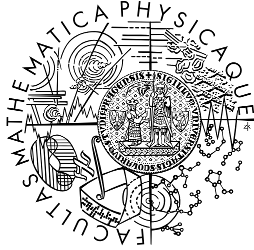


FACULTY OF MATHEMATICS AND PHYSICS
Charles University

MASTER THESIS

Natália Potočeková

Data Lineage Analysis for Databricks Platform

Department of Distributed and Dependable Systems

Supervisor of the master 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 Natália Potočeková
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 .... date ....

Author’s signature

I would like to thank my supervisor, doc. RNDr. Pavel Parízek, Ph.D., for his help, advice, and useful comments whenever I needed anything.
Thanks should also go to RNDr. Lukáš Hermann and Mgr. Lukáš Riedel for their valuable advice regarding Manta Flow platform and ensuring I had all the information needed to work on the thesis assignment.
I'm also extremely grateful to my parents, family, and friends for their support throughout my studies. Without them, it would be very difficult to get this far.

Title: Data Lineage Analysis for Databricks Platform

Author: Natália Potočeková

Department: Department of Distributed and Dependable Systems

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

Abstract: Notebook-based technologies, like Databricks and Jupyter notebooks, have gained popularity in recent years due to their adaptability and convenience. A notebook is an interactive computational environment that allows users to create documents that contain code, visualizations, and explanatory text in one place. Notebooks provide a space for data exploration, analysis, and documentation, enabling users to easily develop and present their work. The ability to combine code execution with explanations and visualizations within a single document promotes reproducibility, enhances collaboration among team members, and motivates data scientists to efficiently work with data. In this work, we analyzed the Databricks technology in order to extend the Manta Flow platform, a highly automated data lineage analysis tool, to support this technology. We designed and implemented a new scanner that provides basic support for analyzing Databricks notebooks written in Python and Databricks SQL languages. We also provide an implementation of a so-called shared context that can be used for passing information between different scanners in the Manta Flow platform. To visualize the interactions between languages and scanners we extended the Manta graph with a new node type that represents the shared context. Alongside this, we implemented a so-called language context to enable scanners to store information useful for analysis of further cells written in the same language in a given notebook. Finally, we demonstrate the functionality of the scanner and the result graphs it produces on example Databricks notebooks.

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

Contents

1 Introduction **3**
1.1 Goals . . . 4
1.2 Glossary . . . 5
1.3 Outline . . . 5

2 Data lineage analysis using Manta Flow **7**
2.1 Manta Flow platform . . . 7
2.2 Python scanner . . . 9
2.3 Embedded code service . . . 12
2.4 SQL scanner . . . 12

3 Databricks **15**
3.1 Data warehouse vs Data lake . . . 15
3.2 Databricks Lakehouse . . . 16
3.3 Key concepts . . . 19
3.4 Notebooks and Queries . . . 21
3.5 External sources . . . 25
3.6 Extracting data from Databricks . . . 25

4 Analysis **34**
4.1 Data necessary for lineage . . . 34
4.2 REST API . . . 35
4.3 Unity Catalog API . . . 40
4.4 Prototype . . . 45
4.5 Hive metastore . . . 46
4.6 Databricks notebooks . . . 50
4.7 Spark context . . . 51
4.8 External sources . . . 54
4.9 Databricks SQL vs Hive SQL . . . 55
4.10 Summary . . . 60

5 Design **62**
5.1 Scanner design . . . 62
5.2 Information extraction . . . 63
5.3 Context information in notebooks . . . 65
5.4 Scanners integration . . . 84
5.5 Notebook analysis . . . 88
5.6 Result graph . . . 99

6 Implementation **101**
6.1 Extractor . . . 101
6.2 Dataflow Generator . . . 102
6.3 Python scanner . . . 109
6.4 Testing . . . 111

7 Evaluation 113
7.1 Databricks SQL and Python interaction example 114
7.2 Python script with Pin nodes example 116
7.3 Limitations and Future Work 116
7.4 Lessons learned 117

8 Conclusion 119

List of Figures 122

A Attachments 123
A.1 User Documentation 123
A.2 Contents of the Attachment 123

1. Introduction

Notebook-based technologies, like Databricks and Jupyter notebooks, have gained popularity in recent years due to their adaptability and convenience. A notebook is an interactive computational environment that allows users to create documents that contain code, visualizations, and explanatory text in one place. Notebooks provide a space for data exploration, analysis, and documentation, enabling users to easily develop and present their work. The ability to combine code execution with explanations and visualizations within a single document promotes reproducibility and enhances collaboration among team members. This flexible and interactive approach to data analysis has contributed to the popularity of notebook-based technologies and motivates data scientists to efficiently work with data and share their findings.

Jupyter Notebook [22] is an open-source web application that can be used by data scientists to create shared documents. These documents can be used for live coding, combining multimedia content, working with data using different languages, and more. However, the downsides of Jupyter Notebook are the difficulty to keep in sync when working collaboratively on a notebook, no way of connecting to the machine learning models, and not good scalability since Jupyter Notebook usually works with data that fit into memory.

That is where Databricks takes place. Databricks [11] is a cloud-based platform built on top of Apache Spark [28] that provides the same functionality as Jupyter Notebook (even provides support for notebooks written in the Jupyter Notebook platform itself) and on top of that provides support for data management, engineering, data science, and machine learning. Databricks provides the Lakehouse Architecture which is available on 3 different clouds: AWS [1], Azure [2], and Google Cloud [3]. Databricks also provides multiple options for storing table and workspace metadata. The most popular options are a built-in Hive metastore or newly a Unity Catalog metastore. With all of this combined Databricks is a powerful tool for managing and working with large amounts of data.

However, what if Databricks is only a small part of the technologies that companies use for working with data? Even though Databricks is really popular nowadays due to its powerful support and tools, it is still relatively new, so many companies are only starting to explore their possibilities with Databricks and while doing so still stick to other technologies, databases and tools for working with data. Also, Databricks is not suitable for every use-case, hence companies can use Databricks as only a part of the technology pool for data management.

Since many important business decisions are being made based on the data that the company has available, it is absolutely necessary to know what is the source of the data and what transformations have been used on them to produce the results.

Imagine that you work in a company that was founded somewhere around the 1980s. In the beginning, they build a simple data model that they used for working with data. However, as years went by, the data model had to be adapted, extended, and adjusted to fit the amount and the structure of data that were produced and obtained each year. By the year 2023, the data model has

been changed many times, main architects that were there at the beginning of the company have already left with their knowledge of the data model and the data that are important for difficult business decisions seem to be odd according to the manager so they ask you to check the data quality and correctness. In the 1980s this would be relatively easy. The only thing needed would be to trace all tables that were used and look at the transformations that have been done with them. After a few days, the problematic transformation or data source would have been discovered and the situation could be fixed easily. However, since we do not live in the 1980s but 2020s the situation is much worse. To manually trace everything would take months and business would suffer since the important decisions would either be made on top of wrong data or would not be done at all due to missing data.

To help with situations like these many companies use so-called cataloging tools to navigate across the data solution. These cataloging tools such as Informatica's Enterprise Data Catalog or IBM's Watson Knowledge Catalog automate the data source discovery throughout the enterprise's systems.

Cataloging is a useful way of navigating through data, however, one important part is missing - the data transformations. For example, the cataloging systems do not show you the origin of the data. In order to do so a data lineage has to be created. Data lineage can be defined as a sort of life cycle that shows the data origin and where it moves over time. Data lineage [9] is useful when trying to analyze how information is used and tracking key bits of information that serve a particular purpose. Over the past few years, several companies managed to create their own data lineage solutions, one of which is the Czech-American company Manta.

Manta started as a project of the Czech company Profinit, however quickly became a separate company selling their own product called Manta Flow. Manta Flow is a complex solution that supports data lineage analysis for over 40 technologies from the fields of business intelligence, data modeling, databases, or programming languages. Thanks to the support of a wide range of technologies (such as different SQL dialects or Python programming language) and the ability to integrate with data cataloging solutions such as Collibra Data Governance Center or Alation Data Catalog, Manta Flow gains a lot of popularity among the enterprise-level customers. However, no notebook technology has been supported by Manta so far. That was the initial motivation for this thesis project.

1.1 Goals

The goal of this thesis project was to extend the Manta Flow platform for automated data lineage analysis with support for Databricks. The design of the scanner and concepts used in it should be applicable to any technology that uses notebooks in a similar manner as Databricks for example Jupyter Notebook. Databricks supports multiple languages in notebooks and the scope of language support in this thesis has been reduced to the Python language and SQL language only. The reason behind this was that scanners for other languages were either not available yet or not sufficient for usage yet. However, the design of the solution had to support a simple addition of a new language scanner so that once the scanners were available they could be used as soon as possible.

The list of specific goals includes:

1. Implementing support for invoking existing scanners for various languages and technologies (Python and SQL).
2. Merging the results of supported technologies into one result graph.
3. Extending the Manta Python scanner to properly handle library procedures and features typically used in Databricks notebooks.
4. Maintaining the shared context (analysis state) between the invocations of different individual scanners (Python and SQL).
5. Implement extraction (loading) of metadata from the respective storage (Hive metastore or Unity catalog).

1.2 Glossary

In this section, we define important terms related to this work.

A *data lineage* is a representation of the relationship between two entities. It usually maps where the data originates from, how it is used and transformed, and where it ends up. Usually, a data lineage is visualized by a graph, however, other visualizations may be used as well.

A *data flow* is a relation of two objects where one is the source or provider of the data and the other is a consumer of provided data.

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

A *Manta scanner* is a part of the Manta Flow platform which is capable of performing data lineage analysis for a specific technology (database management system, ETL tool, or programming language).

A *notebook* is an interface to a document that contains runnable code snippets, narrative text, and visualizations.

1.3 Outline

In this work, as first in Chapter 2 we introduce Manta Flow, and its tools for data lineage computing, describe how to extend the platform to support more technologies, and describe the concepts of tools that are important for this thesis.

Chapter 3, named Databricks, describes the Databricks technology, its main concepts, processes, tools, and data entities.

Chapter 4, named Analysis, defines the scope of this project, then focuses on identifying all features and tools that have to be used and supported in order to compute the lineage for the Databricks platform.

Chapter 5, named Design, discussed the designs of solutions for problems discovered in the Analysis chapter. It provides detailed design for all main concepts necessary for the Databricks platform support in Manta.

Chapter 6, named Implementation, describes how the individual features have been implemented and highlight the implementation details we considered important.

Chapter 7, named Evaluation, demonstrates the functionality of the provided solution, and discusses the limitations of the solution and lessons that we learned when working on this project.

Lastly, Chapter 8, named Conclusion, summarizes the output of this work and compares them to the original goals.

We also provide a Section A, Attachments, at the end of this thesis which contains short user documentation and describes the contents of the attachment provided alongside this work.

2. Data lineage analysis using Manta Flow

Before we get to the main subject of this thesis let us describe how Manta Flow works in general and what parts of the platform we used to create the scanner for the Databricks technology.

2.1 Manta Flow platform

As mentioned above, Manta Flow is a software solution for automated data lineage analysis created by Manta. Its goal is to help users visualize their data pipelines which results in easier data governance, faster data incidents resolution, and improved data quality. The key feature of Manta Flow is automation so the data lineage analysis can be performed in a scope of a few hours for smaller environments or days for huge environments.

Since every technology has a specific way of storing information and different metadata structure, a new Manta scanner has to be developed for each supported technology. It is obvious that the same approach cannot be used for DBMSs such as Oracle and business intelligence tools such as PowerBI.

Thanks to the Manta Flow platform design, the company is capable of abstracting the key concepts present in every technology and developing a new scanner quickly based on the market needs. Currently, Manta supports over 40 technologies from the field of databases, BI, reporting, data modeling and integration, and programming languages.

Every scanner in the Manta Flow platform has the same high-level architecture, which consists of two main components, Connector and Dataflow Generator.

1. **Connector** is further divided into two main parts:
   (a) **Extractor** responsible for extracting (loading) all inputs needed for dataflow analysis to a single location. The extracted information could be stored on some server or other locations specified by users.
   (b) **Reader** responsible for creating a general model used in Dataflow Generator from the extracted data. The model contains interfaces and classes that represent extracted entities in such a form, that the Dataflow Generator and hence the analysis is not dependent on the exact format from extraction. This helps to minimize the changes that have to be made in case the extraction has to change (e.g.: the API endpoint changed the response format a little bit).
2. **Dataflow Generator** which uses the Reader output to create a data lineage graph that can be viewed by users.

Figure 2.1 shows the described scanner architecture.

The result graph generated by the Manta platform contains nodes and edges that represent the lineage and dataflow in scanned environments and programs. For example, in the case of databases, it contains nodes that represent tables,

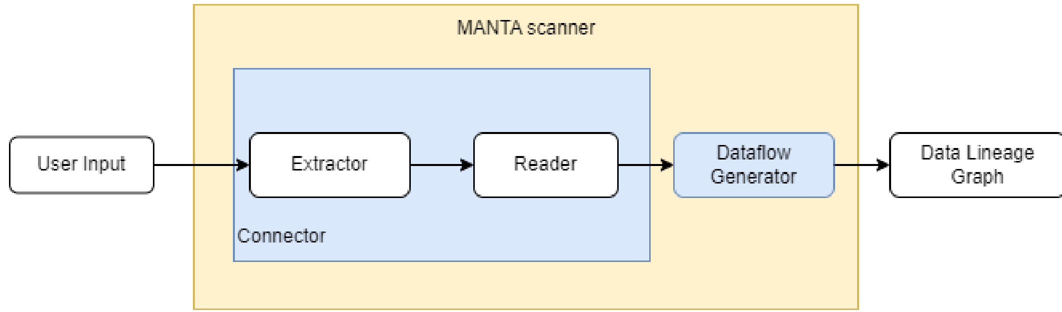


Figure 2.1: High-level scanner architecture diagram

catalogs, or any other database entity stored in the scanned database. For BI tools or programming language scanners, the graph can contain nodes that represent reading/writing from/to standard input or some file, and nodes that represent the analyzed scripts, functions, or modules. The edges between the nodes are oriented and denote the lineage flow between nodes. Figure 2.2 shows an example graph produced by the Manta platform (Python scanner) that denotes the following lineage:

1. Data are read from a CSV file into a pandas Dataframe.
2. The Dataframe is split into two Dataframes. The first one contains the first three columns, and the second contains the last two.
3. The two Dataframes are written into separate files.

Figure 2.2: Example Manta graph from Python scanner

In Manta also a lineage for one specific entity (e.g.: a column) can be depicted by clicking on it. This is visible in Figure 2.2 in blue and yellow colors for the column named `last`.

Now that we understand the overall function and concepts of Manta Flow, let us describe the concrete scanners necessary for Databricks data lineage analysis.

2.2 Python scanner

In order to be able to analyze the parts of the notebooks that are written in Python language the Python scanner [27] is necessary. Let us now describe how the Python scanner works and what it needs in order to create a lineage graph.

The Python source code analysis could be summed up into the following phases:

1. Extraction of code
2. Input processing
3. Call graph construction
4. Computing alias analysis
5. Symbolic data-flow analysis
6. Result graph creation

Let us now briefly describe each phase.

2.2.1 Extraction of code

The first step is collecting all input program codes in a single location. This process is called *extraction*. There are two kinds of source codes that are needed - application and libraries. *Application* is code written and provided by the user and *libraries* are source codes that are used in programs, however, users did not write them themselves. For example, there are built-in libraries available in Python that can be used in any Python application by default and there are many public libraries that can be downloaded and imported by users in their input source codes.

The main goal is to analyze the application code so that users would see the data pipeline of their own programs. However, in order to do so, libraries are necessary since their methods can modify or transform data in a way that can have an impact on lineage. Hence both the application code and libraries need to be included in the analysis.

2.2.2 Input processing

Once all source codes are extracted they need to be loaded into memory and processed. Python scanner processes the source code on its own without any compilation or usage of interpreters. The reason behind this is that this strategy allows bigger control of how the output is represented and hence easier and more precise tailoring of representing data structures. On the other hand, this approach has a downfall that it is not easily compatible with all versions and has to be updated every time a new version of Python is released.

One example of input source code processing is syntactic sugar removal. For example, when we have an expression `'Hello' + 'world'` behind the scenes it is equivalent to an expression `'Hello'.__add__('world')`. In order to be able to analyze the behavior of expressions easier, those that represent the same thing

are transformed into a unified representation. In the provided example it would be the form using the `__add__` method.

2.2.3 Call graph construction

After the code is processed the next thing that needs to be done is the computation of a call graph. The call graph represents the information about what functions can be invoked from a given method (context). Since the callers and callees do not change during the run of an analysis this can be pre-computed once before the analysis begins which saves a lot of valuable time. The same applies to imported modules and the functions they contain. Hence this phase also handles the resolving of imports since functions from imported libraries can be invoked as well.

2.2.4 Computing alias analysis

The next step that needs to be done is to figure out what expressions may reference the same data flow. This is called computing the aliases. This step is important as the scanner needs to correctly assign and propagate data flows in the symbolic analysis. Let us now show a simple example of the aliasing:

```
1 foo = 'Hello world'  # foo aliases the value 'Hello world'
2 bar = foo            # bar also aliases the value 'Hello world'
```

Aliases are computed per different parts of the input source code such as functions, classes, and modules.

2.2.5 Symbolic data-flow analysis

After the computing alias analysis, the symbolic analysis can take place. Symbolic analysis means that the input source code is analyzed without running the code before, hence no runtime information is available. For example when we have branching in the input source code using the `if ... else ...` statement the analysis has no way of knowing which branch was actually used in runtime. Hence it needs to consider both options.

The analysis starts on a so-called *entry point* which represents a function or a file that is invoked first when the program is run by an interpreter. During the run of symbolic analysis, all objects that need to be analyzed (functions, classes, and modules) are put into a worklist.

The worklist algorithm processes the objects in the worklist one by one until a fixed state over the dataflows is reached. That means that the algorithm stops when there are no new flows created. In the Python scanner, the worklist contains all objects reachable from the entry point at the beginning of the algorithm. To represent a place where the function was invoked a so-called *Invocation context* is used. The invocation context represents the function or method and the flows associated with its arguments. When the analysis of a single function or method is finished, the algorithm checks if the flows changed in any way. If yes, the function/method and all its callers and callees need to be added back to the worklist to be analyzed in the next iteration.

During the run of the analysis for a single function or method, the analysis keeps a set of tracked expressions. In the beginning, the arguments of the function/method are tracked. Other expressions that can be tracked later are assignments and function/method calls. In the case of assignments, the left side of the assignment is tracked and flows associated with the right side are *propagated* to the left side expression. This is similar for function or method calls. For example, when we pass the variable `foo` into the function as a first argument, the flows associated with the variable will be propagated to the first argument of the function. However, that means that the flows will be stored in the invocation context of the called method rather then in the invocation context of the current method.

The method calls and assignments we mentioned above construct a so-called *flow summary*. The flow summary represents the flow data for a particular invocation context. These flow summaries are then used to get the final result of the analysis.

In order to speed up the execution of the analysis few approximations have to be made. For example, the code from the libraries is not simulated. Instead, the Python scanner contains so-called *propagation modes*. These propagation modes are written for those library functions that have an impact on the lineage. The propagation modes handle the data flow in a special way that is specific for each function.

For example, when we use the Pandas method `read_csv` the propagation mode takes all of the `UnknownResourceColumnFlows` associated with the source (the reason behind this is that there is no way to know precisely what columns are present in the file since only symbolic analysis is performed), for each such flow creates new `PandasColumnFlow` and registers it to the propagation target. By doing this, the scanner preserves information that there were data read from the CSV file and stored in the Pandas dataframe. Later, when there are some transformations done with the data, the flows created here are used, hence the lineage keeps extending.

Using the modifications the analysis is faster and can finish in a reasonable time. However, the price for it is that the flows that can be tracked are limited by the propagation modes that are implemented. All other information that is not in the user input source code or propagation modes is lost.

2.2.6 Result graph creation

After the analysis is finished the results are transformed to a standard Manta graph. That means that nodes and edges are added for those flows that are important for users to see such as file and database streams. This transformation is done in the common Dataflow Generator for all intermediate languages supported in Manta currently. This can be done since the output of the Connector does not contain any language-specific information and hence the forward process can be the same for all intermediate languages. After this phase, the graph is ready to be visualized to the users.

2.3 Embedded code service

Another desired feature of Manta (Python scanner) is the ability to run the scanner on small code snippets embedded in different technologies (e.g.: Databricks). To do so, the Embedded code service has been used.

Embedded code service (or ECS for short) is a standalone service for the analysis of an intermediate language code embedded in a different technology. Each intermediate language has its own ECS that focuses on the language scanner requirements. In our case, we will use the Python ECS.

The Python ECS is responsible for properly setting up the input for the Python scanner, calling the Python scanner analysis, and then returning results to the caller technology. Let us now describe this process in more detail.

The initial phase is the *Initialization*. This phase is responsible for initializing and preparing graph for Python analysis. Then the *Orchestration* phase takes place. This phase prepares a temporary directory where the input for the analysis is put. Then each technology has its own *Orchestrator* that is responsible for properly preparing all files that should be extracted and defining an entry point. Then the extraction is executed and the configuration for analysis is prepared. Next, the Python analysis is executed based on the configuration obtained in previous phases. When the analysis finishes the result is transformed to a Manta graph. Before the final results are returned the *Cleanup* phase takes place and all data in the temporary directory and the directory itself are deleted. Then the result graph obtained from the Python scanner is wrapped to a structure that represents the result of the embedded code service and provides an interface for working with the Python result graph such as merging it to a specified parent graph or management of PIN nodes.

Figure 2.3 shows the high-level image of the ECS phases.

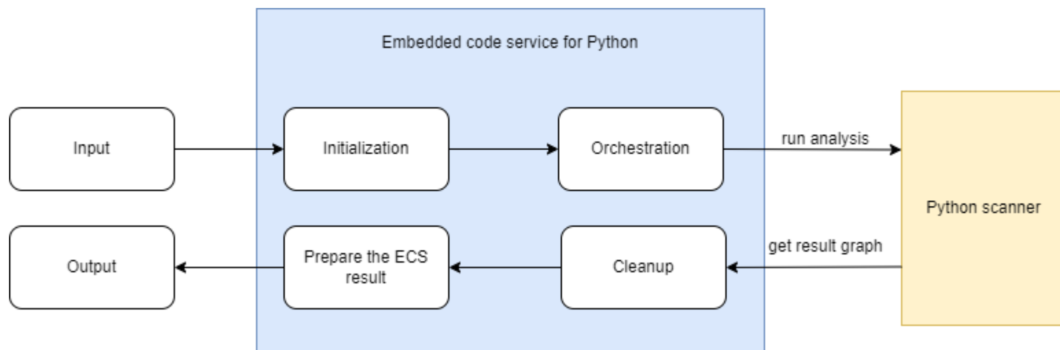


Figure 2.3: ECS phases high-level diagram

2.4 SQL scanner

In order to be able to analyze the SQL parts of notebooks we will need to use an SQL scanner. In this section, we describe what a standard SQL scanner looks like and what is each component's function in order to create a lineage graph.

The standard SQL scanner has two major components - the Connector and Dataflow Generator. The Connector is then divided into Extractor, Dictionary, Model, and Resolver. The Extractor is responsible for extracting information

from a given database. The information can be for example DDL scripts, however, users can also manually provide scripts alongside the extraction. Then the Dictionary contains, for each identifier, some information about what it represents based on the hierarchy of entities in the database. For example tables, variables, datatype, builtin-functions, and more. The dictionary keeps all of this information in one place so that it would be easier to work with them in further phases. The Model component defines the abstract syntax tree (or AST for short). The AST is then analyzed in the Resolver component. Resolver contains the parsing and semantic analysis of the AST. Then the data are passed to the Dataflow Generator. Dataflow Generator represents a bridge between Connector and Manta Flow platform itself. Its task is to transform the output of the Connector to the common intermediate language structures that will be visualized in the final graph. In other words, for each statement important from the lineage point of view the Dataflow Generator produces source and target nodes and connects them.

Now that we know the data flow in the SQL scanner let us describe it in more detail. As we already mentioned, the Extractor component extracts the content of the database and as a result, the scanner gets DDLs and the Dictionary. The DDLs are then passed for language processing which has in the SQL scanners four phases - lexical analysis, syntax analysis, semantic analysis, and further processing. The Lexer component is responsible for the lexical analysis which means that it takes the input DDL and parses it into tokens or lexemes (identifiers, keywords, literals, and special symbols). Then the input syntax is analyzed in the Parser component. This produces the AST tree usually slightly modified to suit the analyzed technology. Then the AST is passed to the Resolver component where the semantic analysis takes place. Semantic analysis analyses the meaning of statements. For example, it answers the question what does `a` mean in the `SELECT a FROM t1 CROSS JOIN t2`. It could be a column from `t1` or `t2`, a variable or any other construct. The semantic analysis gives answers to such questions. Once all such questions are answered, further processing takes place. In the case of the SQL scanners, further processing means that the data flow analysis takes place in the Dataflow component. This component then produces a graph that is passed to a Post-processing module. After that, the final graph is ready. Figure 2.4 shows the whole workflow.

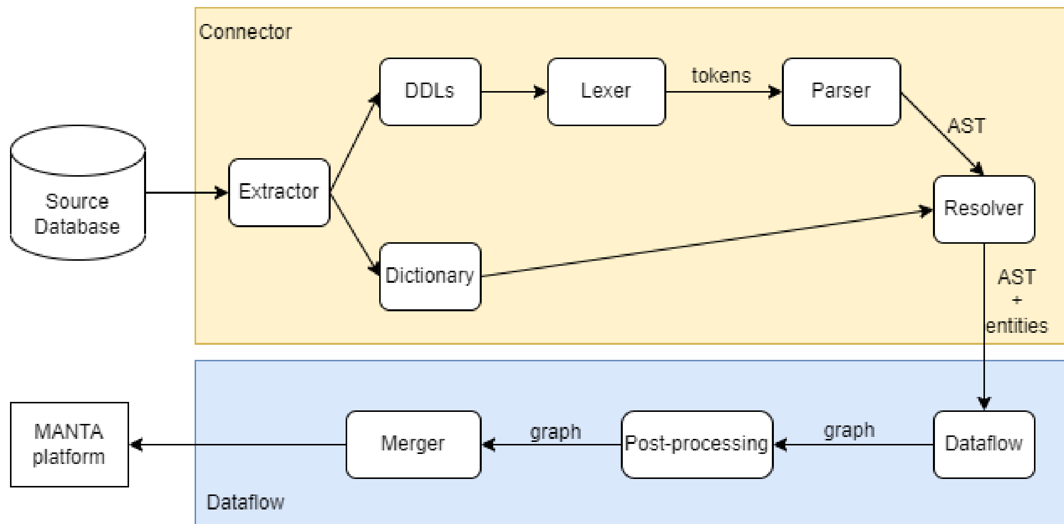


Figure 2.4: SQL scanner phases high-level diagram

3. Databricks

Databricks is a cloud-based platform built on top of Spark [28] used for data science, engineering, and analytics. Databricks combines the advantages of data warehouses and data lakes into so-called lakehouse architecture.

3.1 Data warehouse vs Data lake

Both data warehouse and data lake are used as storage for large amounts of data. **Data warehouse** [10] stores these data in a structured manner in files and folders so that data are available for reporting and analysis. In order to store data in a warehouse a schema has to be defined first. Then anytime data are stored in a warehouse they are formatted accordingly to fit all data that were stored there before. This allows data warehouses to provide highly performant and scalable analytics which is useful for BI and reporting. However, the disadvantage of this approach is that it is more expensive management-wise, and cannot process some kinds of data. Hence there is no support for modern data science techniques and machine learning which require data such as images and videos.

On the other hand, **data lake** [8] stores data in its raw format using a flat architecture and object storage. A data lake is used when the schema is not known or specified upfront, when all data are supposed to be in a single central location, or when unstructured data needs to be processed. Any data type can be processed in a data lake, even semi-structured and unstructured data such as images, videos, and audio. Due to this data lakes can be used for machine learning, or advanced analytics which are commonly used in today's data science. However, since the data are not structured in any way it is harder for data lakes to maintain reliable data which can lead to inaccurate query results, poor performance, and poor BI support.

In order to get the best of both worlds a combined approach is usually used which is called Lakehouse.

Data lakehouse starts as a data lake that contains all data types and is later converted to Delta Lake format which enables ACID transactions [5] from traditional data warehouses on data lakes.

Key attributes of a lakehouse are:

- Transaction support
- Schema enforcement and governance
- BI support
- Storage is decoupled from computation
- Openness
- Support for diverse data types ranging from unstructured to structured data
- Support for diverse workloads
- End-to-end streaming

The difference between the data warehouse, data lake, and data lakehouse is shown in Figure 3.1 which was taken from the article describing the differences of given architectures [30].

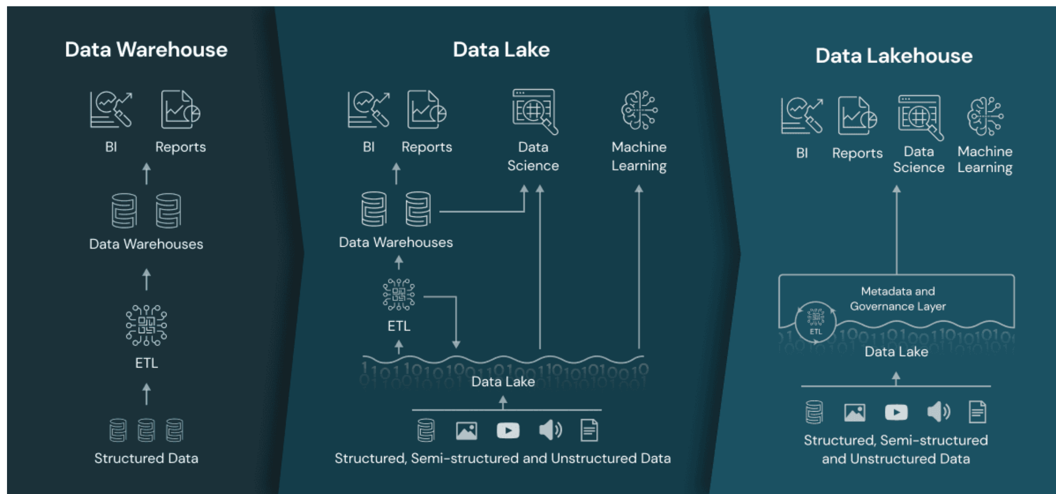


Figure 3.1: Difference between data warehouse, lake, and lakehouse [30]

Let us now describe technologies and concepts used in Databricks in more detail. To be more precise the Delta Lake in Section 3.2.1, Databricks Lakehouse in Section 3.2, Key concepts in Section 3.3. Then we dive more into detail about notebooks and queries in Section 3.4, external sources in Section 3.5, and last but not least the tools that can be used for extracting data in Section 3.6.

3.2 Databricks Lakehouse

As we mentioned in the previous section, lakehouse combines the advantages of data warehouses and data lakes. In Databricks, the implementation of the lakehouse is called **Databricks Lakehouse**.

Databricks Lakehouse provides ACID transactions and data governance similarly to data warehouses and combines it with the flexibility and cost-efficiency of data lakes. This is used to enable business intelligence and machine learning on data. Data are stored in a scalable cloud object storage in open source data standards.

Primarily, lakehouse consists of two components: Delta tables and Unity Catalog.

3.2.1 Delta Lake and Delta Tables

One of the most important concepts for platforms that work with data is the storage layer. The storage layer is responsible for storing all data (e.g.: tables) available in the platform in a pre-defined manner. In the Databricks Lakehouse platform, the storage layer is called **Delta Lake** [29].

Delta Lake is an open-source software and it is an extension of the Parquet data files with a file-based transaction log for ACID transactions and metadata

handling. It is fully compatible with Apache Spark APIs and it is used as a default storage format for all operations in Databricks. If not specified otherwise, all tables are stored as **Delta tables**. That means that when a new table is created following actions take place:

1. the metadata used to reference the table is saved to the metastore in the declared schema or database

2. both, data and metadata are saved to a folder (in Parquet format) in the object cloud storage

Now that we know the high-level overview of the platform's architecture, let us describe the platform in more detail. In the following sections, we describe what is metastore, what features it has, and the different kinds of metastores that Databricks provides.

3.2.2 Metastore

Metastore is a top-level container that stores tables and views and permissions for accessing data. Only admins can create metastores and assign roles and permissions in the metastore to other users.

There are three options for metastore:

- Unity Catalog
- Hive metastore
- External metastore (legacy)
  - External Apache Hive metastore
  - AWS Glue Data Catalog

Previous versions of Databricks worked mostly with Hive metastore, however, Unity Catalog provides a new metastore with better security and auditing.

Each metastore has a default storage location for managed tables in Amazon S3. This storage is accessed with storage credentials that were created during the metastore creation. In order to assure higher security, user code does not have access to the credentials directly but Unity Catalog generates an access token for each user or application separately so that they would be able to access requested data.

External tables are stored in S3 as well, but in other paths and they can be accessed by additional storage credentials added by admin.

Figure 3.2 shows the object model in the Unity catalog. This diagram was created based on the article "What is Unity Catalog?" [31].

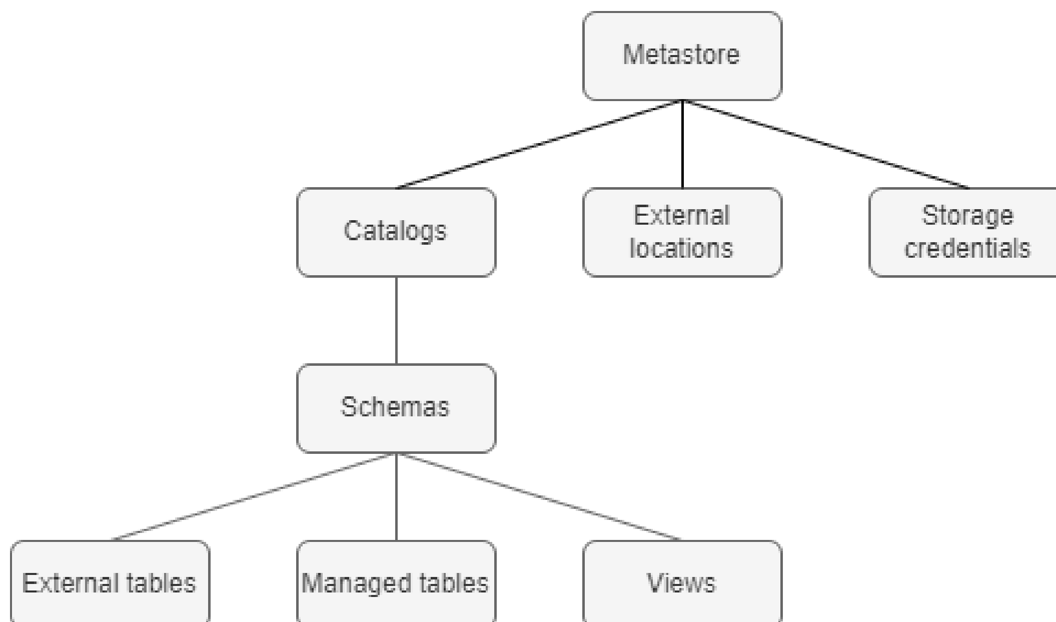


Figure 3.2: The object model structure in Unity Catalog

Since Unity Catalog is currently the preferred metastore from Databricks point of view, let us describe it in more detail.

Namespace

Unity Catalog organizes data in the so-called three-level namespace. The three layers are Catalog, Schema, and Table.

Catalog is the highest layer and is used for organizing data assets. Users can access the catalog only when USAGE data permissions are correctly set for them.

Schema is the second layer in the namespace and is used for organizing tables and views. Sometimes schemas are also called databases. Data in the schema can be accessed only with USAGE data permissions for the schema itself and SELECT data permissions for a given table or view.

The third layer of the namespace is dedicated to *tables* and *views* which contain rows of data. Tables can be created only with CREATE and USAGE permissions for the parent schema and USAGE permissions for the parent catalog. To query the table the SELECT permissions on the table are needed alongside USAGE permissions on its parent schema and catalog.

There are two kinds of tables - managed and external.

Managed table is a default way to create a table and is stored in the managed storage location which was configured during metastore creation. These tables are managed in the Delta format only.

External table is stored outside the managed stored location and is not fully managed by Unity Catalog. These tables can use the following formats: Delta, CSV, JSON, AVRO, Parquet, ORC, and text.

In order to manage access to external tables, there are two object types used in the Unity Catalog - storage credentials and external location. *Storage credential* is an authorization and authentication mechanism for accessing data on the cloud. An *external location* contains a reference to a storage credential and path on cloud storage (it grants access only to the path and its child directories/files).

Alongside managed and external tables, views are in the third layer of the namespace. They are read-only objects and are composed from one or multiple tables and views in metastore (they can be in different schemas or catalogs). Views are created by the SQL command `CREATE VIEW` in a standard way.

Cluster security mode

Now that we know the basic structure of the metastore another thing we need to take a look at is the security aspect as well. In order to guarantee isolation and access control Unity Catalog uses a so-called security mode. In order for the cluster to access the Unity Catalog, it needs to have the appropriate security mode configured (this means that the High Concurrency cluster is not available in Unity Catalog).

Types of security modes:

- None
  - no isolation
- Sigle user
  - by default the person that created the cluster (can be transferred to another user)
- User isolation
  - multiple users, only SQL workloads supported
- Table ACL only
  - workspace - local table access control
- Passthrough only
  - workspace - local credential passthrough

3.3 Key concepts

Now that we know what Databricks is and what entities it used for storing data, let us introduce the key concepts used in Databricks.

3.3.1 Workspace and account

The term *workspace* can have two meanings in Databricks.

1. Workspace is an environment deployed in the cloud that is used for accessing Databricks assets. Organizations can have one or multiple workspaces depending on their needs and business logic.
2. Workspace is the User interface (UI) for Data Science & Engineering and Machine Learning environments.

The *workspace browser* is the UI that lets users browse their notebooks, jobs, libraries, and other files in the Data Science & Engineering and Machine Learning environments.

A *Databricks account* is a single entity used for billing purposes or support. One account can include multiple workspaces.

3.3.2 Authentication and authorization

Since Databricks is a cloud-based platform, authentication and authorization policies have to be provided to ensure data security. The following terms and concepts are used in Databricks to do so.

User is an individual that has access to resources and the system. Users are identified by their email address.

Service principal is a service identity that is used in the context of jobs, scripts, automated tools, and CI/CD platforms. The service principal is represented by an application ID.

Group is a collection of users and is used for easier identification and access management.

The *Access Control List* (ACL) is a list of permissions for workspace, cluster, job, experiment or table. The ACL specifies what users can access which objects and what operations they can do with them.

The *Personal Access Token* is an opaque string constant used for authentication in the REST API and in tools that connect to the SQL warehouses in Databricks.

3.3.3 Data Science and Engineering

Another feature that Databricks provides is the environment, where data scientists can work together, and share their knowledge, data, and results. Exactly for this purpose the Data Science & Engineering environment is provided by Databricks.

Data Science & Engineering is an environment used for collaboration among data engineers, data scientists, and data analysts. The following concepts are used in this environment to ensure an understandable, easy, and intuitive work with the environment itself and its entities. The entities users can work with are:

1. Notebooks
2. Dashboard
3. Library
4. Repo
5. Experiment

Notebook is an interface to a document that contains runnable code snippets, narrative text, and visualizations. (More on notebooks in Section 3.4).

Dashboard is used for publishing graphs and visualizations from notebook outputs and sharing them in a presentation format. Usually, a special job is created for updating the dashboard.

Library in Databricks contexts is basically the same thing as libraries in programming languages - a package of code that is available to notebooks or jobs running on a cluster. There are "builtin" libraries that are present for all runtimes and also own libraries can be added if needed.

Repo is a folder that has a co-versioned folder content with some remote git repository.

Expetiment is a collection of machine learning training runs.

The data assets can be accessed by following tools:

1. User interface (UI)
2. REST API (To see the REST API analysis please see Section 4.2)
3. Command-line interface (CLI)

For data management, the following concepts are used:

1. Databricks File System (DBFS)
2. Database
3. Table
4. Metastore

The *Databricks File System* (DBFS) is an abstraction layer over blob storage that contains folders and files (libraries, data files, images). In the beginning, this storage contains training datasets that can be used for learning how to work with Databricks.

3.4 Notebooks and Queries

When working with data we have several options for how to do so. The first option is to create a Databricks **notebook**. Databricks notebooks consist of so-called **cells**. Each cell can be written in a different language that is supported in Databricks. In order to execute the notebook code, the notebook is attached to a cluster and then the cluster runs all the cells in the notebook in the order they were written. Notebooks can be either run manually or the user can schedule a so-called **job** that will execute the notebook automatically based on the job parameters.

Another way to work with data is to create a **query**. Query is a simple SQL script that can be run on a cluster and for simplicity, we can imagine it as if it was a notebook with only one cell written in the SQL language.

Now that we know what notebooks are from a high-level point of view, let us describe them in more detail.

3.4.1 Language Magic Command

Since notebooks enable writing the cell content in different languages, there has to be a way how to distinguish which language is being used.

In Databricks these languages are enabled:

- Markdown
- Python (default for cell usually)
- SQL
- Scala
- R

When using any of these supported languages, the Language magic command is used to define which language is being used. The following table contains language and its commands:

Language	Magic command
Markdown	%md
Python	%python
SQL	%sql
Scala	%scala
R	%r

3.4.2 Spark context

As we know from the previous section, Databricks notebooks support the following languages for cells: Python, Scala, R, SQL, and Markdown. Out of the mentioned languages, only Markdown does not execute any code. Since notebooks can contain cells in different languages we need to take a deeper look at what information cells in different languages can pass to one another. In the following sections, we describe how this can be done in Databricks.

Basic overview

According to the article about working with multiple languages on Databricks [17] each language behaves as if it was in its separate virtual machine. That means that by default languages do not see each other's variables. However, there is one option that can be used to change that. Since Databricks runs on Apache Spark a so-called spark session is always created for each notebook. Using this spark session, languages can share variable values and dataframes.

According to the article about different ways of passing data between languages [13], it is also possible to pass values using so-called widgets. However, this solution only works when executing cells one by one interactively [6].

Since Manta performs static analysis of the code and hence scanners do not execute the code, we focus on the spark session option from this moment further.

Spark session

As mentioned before, the most recommended way for sharing the variables and their values is using the spark session since it is the only thing that is shared among all languages. That means that if one language changes anything in the context, it shows in all languages. This can be useful when we want to perform multiple operations in a row but each of them would be easier to do in a different language. For example, Python is great for accessing files and loading data from them, however, SQL is better when creating tables and views. And for example, R is better when we want to compute some statistics. Using this scenario, the data that the languages would share could be like this:

1. Python cell loads data from the file and stores the dataframe and table name into the spark context.
2. SQL cell uses the table name to access the data loaded by the Python cell and based on some conditions returns a filtered set of results from the original data.
3. R cell takes the latest SQL result and performs some statistical methods on them.

Even though this is a very simple example it demonstrates greatly why using a spark session is useful and hence why we need to be able to handle this concept properly in our scanner. A thorough analysis of what kinds of operations can be used to pass data using the shared context can be found in Section 4.7

3.4.3 Export

Databricks also provides different ways of exporting notebooks. Let us now describe the formats in which the notebooks can be exported. This description is a base for the analysis in Section 4.6.1 that discusses the best format for our scanner.

DBC Archive

The first option for a single notebook export is a DBC archive.

DBC archive is a binary format that includes metadata and notebook command results. According to the Databricks documentation, it is a JAR file with some sort of special metadata added and stored with the extension `.dbc`.

Source File

This option downloads the notebook as `.scala`, `.py`, `.sql`, or `.r` file. The language of the file is determined based on the default notebook language. As an example, we exported a simple notebook that contained Markdown, SQL, and Python cells. The content of the file was in the following format:

```
# Databricks notebook source
# MAGIC %md
# MAGIC Following cell should create spark table from CSV data:

# COMMAND ----------

# MAGIC %sql
```

```
8  # MAGIC DROP TABLE IF EXISTS diamonds;
9  # MAGIC
10 # MAGIC CREATE TABLE diamonds USING CSV OPTIONS (path "/
     databricks-datasets/Rdatasets/data-001/csv/ggplot2/diamonds.
     csv", header "true")
11
12 # COMMAND ----------
13
14 # MAGIC %md
15 # MAGIC Following cells should create Delta table stored at the
     specified location.
16
17 # COMMAND ----------
18
19 # MAGIC %python
20 # MAGIC
21 # MAGIC diamonds = (spark.read
22 # MAGIC   .format("csv")
23 # MAGIC   .option("header", "true")
24 # MAGIC   .option("inferSchema", "true")
25 # MAGIC   .load("/databricks-datasets/Rdatasets/data-001/csv/
     ggplot2/diamonds.csv")
26 # MAGIC )
27 # MAGIC
28 # MAGIC diamonds.write.format("delta").save("/mnt/delta/diamonds
     ")
29
30 # COMMAND ----------
31
32 # MAGIC %sql
33 # MAGIC DROP TABLE IF EXISTS diamonds;
34 # MAGIC
35 # MAGIC CREATE TABLE diamonds USING DELTA LOCATION '/mnt/delta/
     diamonds/'
36
37 # COMMAND ----------
38
39 # MAGIC %sql
40 # MAGIC SELECT color, avg(price) AS price FROM diamonds GROUP BY
     color ORDER BY COLOR
```

As can be seen in the snippet above, each cell is represented by a set of `# MAGIC` comments and are divided by the `# COMMAND ----------` comments.

IPython Notebook

Another option for notebooks export is the IPython notebook file. It is a Jupyter notebook with the extension `.ipynb`.

HTML

The last option for notebooks export is the HTML file. The notebook is converted to a standard HTML source code alongside lots of javascript functions to preserve the notebook's functionality properly.

3.5 External sources

Alongside creating and storing the data in Databricks, there is also an option to interact with external sources. As can be seen in the Databricks documentation [19], both loading and storing from/to external sources can be done. Figure 3.3 shows the relationships between systems.

Figure 3.3: Databricks interaction with external sources

There are multiple formats and source and consumer technologies that are supported in Databricks. Since there are many supported technologies, we mention only a few most important in our opinion. To see the full list please see the official documentation [19].

Regarding the formats, for example, the following data formats are supported:

- Delta Lake
- Parquet
- JSON
- CSV
- Binary
- Text

Databricks can also interact with messaging services like Kafka, or Kinesis that are streaming from or to the message queues. Another possible way of interacting with external technologies is to use the JDBC connection. In this way, Databricks can connect with for example PostgreSQL, MySQL, Oracle, or MariaDB. The last option we mention is integration with data services like Google BigQuery, MongoDB, Cassandra, or ElasticSearch. Integration with these services is done using the connection settings, networking settings, and security credentials.

3.6 Extracting data from Databricks

Now that we have a better image of Databricks and the key concepts used there, let us describe the main tools available for extracting data. There are three most used tools - REST API, Unity Catalog API, and JDBC connection. We describe each of the mentioned tools in the sections below.

3.6.1 REST API

For obtaining information about clusters, notebooks, jobs, queries or workspace as a whole, Databricks provides their REST API. In this section, we describe those methods from the REST API that were suitable for our use case regarding the extraction of data. To be more precise, we describe the groups of endpoints and what their purpose is. Then later in Section 4.2 we describe those endpoints that have useful information for extraction from the data lineage computation point of view.

DBFS API 2.0

One of the useful information we could try to obtain is the information about data sources stored in the file system. For this purpose, the Databricks REST API provides a group of endpoints called *DBFS API 2.0*. This part of the API interacts with different data sources without the need of including credentials every time. However, it cannot be used with firewall-enabled storage. In the following sections, we describe concrete endpoints that can be used to obtain the information we need, the request we used, and the example response we got.

Get status

In order to receive information about a file or directory a request on endpoint `/api/2.0/dbfs/get-status` has to be made with the path of the resource as a request parameter. In case the resource does not exist a `RESOURCE_DOES_NOT_-EXIST` exception is thrown.

The following code sample shows the request structure we used to obtain the information:

```
curl --netrc -X GET \
https://<INSTANCE_URL>/api/2.0/dbfs/get-status \
--data '{ "path": "/tmp/HelloWorld.txt" }' \
| jq .
```

And the following code sample shows the response we obtained:

```
{
  "path": "/tmp/HelloWorld.txt",
  "is_dir": false,
  "file_size": 13,
  "modification_time": 1622054945000
}
```

List

Another useful action is listing the contents of directories or getting more details about files. In order to do so a request on endpoint `/api/2.0/dbfs/list` has to be made. The following code represents a request structure we used to test the endpoint.

```
curl --netrc -X GET \
https://<INSTANCE_URL>/api/2.0/dbfs/list \
--data '{ "path": "/tmp" }' \
| jq .
```

After the request, we obtained the following response. Note that due to the fact that the response was really long, we show only the first object fully, then the rest is replaced by the three-dot notation. However, all of the objects present in the response have the same attributes available.

```
{
  "files": [
    {
      "path": "/tmp/HelloWorld.txt",
      "is_dir": false,
      "file_size": 13,
      "modification_time": 1622054945000
    },
    {
      ...
    }
  ]
}
```

As we can see, the information we obtain about files listed in the directory is the same as the information described in the previous section that discussed the data source details endpoint. The timeout for this method is approximately 60 seconds so only dictionaries with less than 10 000 files can be processed which is a slight limitation.

Read

Now that we know how to get the list of all files present in the directory, we need a way to get to their content. To do so, the REST API provides a read method that returns the contents of a specified file. Maximal size is 1MB and when exceeded a `MAX_READ_SIZE_EXCEEDED` exception is thrown. If the file does not exist a `RESOURCE_DOES_NOT_EXIST` exception is thrown. In case the specified path is a directory, the offset is negative, or the length is negative an `INVALID_PARAMETER_VALUE` exception is thrown.

The following code represents the request structure we used to test the endpoint:

```
curl --netrc -X GET \
https://<INSTANCE_URL>/api/2.0/dbfs/read \
--data '{ "path": "/tmp/HelloWorld.txt", "offset": 1, "length": 8
    }' \
| jq .
```

We obtained the following response:

```
{
  "bytes_read": 8,
  "data": "ZWxsbywgV28="
}
```

Workspace API 2.0

As we mentioned in Section 3.4 notebooks are one of the main ways to work with data in Databricks. Since they can perform transformations with data, create new data from the existing ones or simply reorganize the data structure,

we need to analyze their content in order to produce a lineage graph. In order to get to the notebook contents, the REST API section called *Workspace API 2.0* can be used. This group contains endpoints for notebooks and all related actions regarding them. The maximum size per request is 10 MB. From this API we cannot use import or delete functions since we cannot change customer's data. In the following sections, we describe useful methods that can be used for extracting notebooks.

List

The first useful action that comes to mind is listing all available notebooks in a specified directory or the whole workspace (which is represented by the root directory). In order to do so, the REST API provides the GET method for listing all contents of the workspace on the `api/2.0/workspace/list` endpoint.

In order to get to the notebook content request has to contain the authorization token. In case the token does not provide sufficient access rights, the request fails. The following code represents the request structure we used to test the endpoint.

```
curl -X GET --header "Authorization: Bearer $DATABRICKS_TOKEN" \
https://<INSTANCE_URL>/api/2.0/workspace/list \
--data '{ "path": "/Users/WORKSPACE_NAME" }'
```

For the request, we obtained the following response:


```
{
    "objects": [
        {
            "object_type": "NOTEBOOK",
            "path": "/Users/WORKSPACE_NAME/Data Lineage Test",
            "language": "PYTHON",
            "object_id": 754855114107838
        },
        {
            "object_type": "NOTEBOOK",
            "path": "/Users/WORKSPACE_NAME/UCtest",
            "language": "PYTHON",
            "object_id": 1582990291900299
        },
        {
            "object_type": "NOTEBOOK",
            "path": "/Users/WORKSPACE_NAME/anotherTest",
            "language": "PYTHON",
            "object_id": 1582990291900320
        }
    ]
}
```


Export

Once we know the paths and names of notebooks in the directory or workspace, we need to export their contents. For this purpose the export method on endpoint `/api/2.0/workspace/export` is present. Let us now describe to what formats the notebooks and directories can be exported.

When exporting notebooks one of the following four options can be selected (please see Section 3.4.3 for a detailed description of the formats):

- SOURCE - exported as source code in the main language of the notebook
- HTML
- JUPYTER - Jupyter notebook format
- DBC - Databricks archive format

A directory can be exported only in `dbc` format.

To test the endpoint we used the request structure shown in the code sample below. The parameters of the request were the path to a notebook, the format of the export set to the `SOURCE` option and we also enabled the direct download option so that the file would be stored in the file system.

```
curl --netrc --request GET \
  https://<INSTANCE_URL>/api/2.0/workspace/export \
  --header 'Accept: application/json' \
  --data '{ "path": "/Users/me@example.com/MyFolder/MyNotebook",
   "format": "SOURCE", "direct_download": true }'
```

When downloading notebook content there are two possible download methods:

- direct download - can be done only if the proper flag is set to true: `"direct_download": true`, file is then directly downloaded to the file system
- base64 in JSON - when `"direct_download": false`

The following example shows the download to a base64 format.


```
{
  "content": "Ly8gRGF0YWJyaWNrcyBub3RlYm9vayBzb3VyY2UKMSsx",
}
```


Jobs API 2.1

Now that we know how to export the notebooks from the Databricks workspace, we need to take a look at how to identify all the jobs that can be used to automatically run the notebooks. To do so, the REST API provides a whole group of endpoints called *Jobs API 2.1*. All of the methods available can be found on the official documentation page [21].

However, we describe only two endpoints that we considered useful. The first endpoint is for listing all jobs. The method is available on the `/2.1/jobs/list` endpoint. We decided to skip the example response in this case, since the example response was too long. However, the example response can be found in the documentation [21]. Later, in Section 4.2 we show an example response with only such information that we considered useful for lineage.

The second endpoint we would like to mention is the endpoint used to obtain information about a single job. The method is available in the `/2.1/jobs/get/{job_id}` endpoint.

Queries API

Another part of the endpoints we describe are the endpoints that are used to obtain information about queries. Queries are SQL scripts stored in the data warehouses in Databricks and can be used in a similar fashion as jobs for notebooks. Hence these queries can be viewed as SQL scripts we want to extract and then analyze to see the lineage.

In order to retrieve information about queries, a request on endpoint `/api/2.0/preview/sql/queries` has to be made. This endpoint returns a list of all queries and information about them. Here again, we decided to skip an example response since it was too long. The complete full response can be found in the Documentation [15] and an example of the response with only information useful for lineage computation can be found in Section 4.2.

Data sources

In case we would like to see the list of all available warehouses a GET method on endpoint `/api/2.0/preview/sql/data_sources` has to be called. The response obtained from such a request would look as follows:


```
1  [
2    {
3      "id": "f7df1dfd-565d-4506-accb-8a1e0f8fad09",
4      "name": "starter-warehouse",
5      "pause_reason": null,
6      "paused": 0,
7      "supports_auto_limit": true,
8      "syntax": "sql",
9      "type": "databricks_internal",
10     "view_only": false,
11     "warehouse_id": "3d939b0cc668be06"
12   }
13 ]
```


3.6.2 Unity Catalog API

When Databricks announced the Unity Catalog preview, they also created the data lineage REST API. This API is an extension of the REST API mentioned in the previous section, however, the methods present in the Unity Catalog version are available only for the instances that have Unity Catalog enabled. In case the Databricks instance uses still Hive metastore as a primary storage another method has to be used. This situation is analyzed in Section 4.5. Let us now take a look at the Unity Catalog API extension and the method it provides for data retrieval.

For all following endpoints, the common prefix is: `/api/2.0/unity-catalog`

Metastores and External Locations

If we want to be able to download data information from all places users use in their Databricks instance we might need to list all of the used metastores first.

When it comes to the information about metastores there are two methods that can be called. The first one gathers information about all metastores used

by the instance, the second one gathers information from one specific metastore based on the ID.

To obtain the list of all available metastores the `<prefix>/metastores` endpoint has to be used. We decided not to show the full request response, since it was too long. However, we show an example response in Section 4.3 where we analyze what parts of the obtained information are useful for data lineage computation.

It is possible to retrieve information about external location storage as well. In order to do so following endpoints can be used:

Method	URI	Endpoint name	Function
GET	`<prefix>/external-locations`	listExternalLocations	returns array of External-Location-Info
GET	`<prefix>/external-locations/:name`	getExternalLocation	returns External-Location-Info
GET	`<prefix>/files`	listFiles	returns array of file information

Catalogs

As mentioned in Section 3.2.2 there is a three-level namespace in Databricks. The topmost level is the catalog. Hence the first step in order to get table and lineage information is to get all available catalogs, since tables need to be referred to using their full name in a form `catalog.schema.table`. In order to list all catalogs in the metastore the `/api/2.0/unity-catalog/catalogs` endpoint can be used.

The following code snippet shows an example request we used to test the endpoint.

```
1 curl -n -v -X GET --header "Authorization: Bearer
    $DATABRICKS_TOKEN" \
2 -H 'Content-Type: application/json' https://DATABRICKS_INSTANCE/
    api/2.0/unity-catalog/catalogs
```

We decided to skip the example response in this case since it was too long. We will show an example response with only the information necessary for the lineage computation in Section 4.3. In order to see the full list of fields that are returned please see the documentation [23].

There is also an option to get information about a single catalog through the endpoint `<prefix>/catalogs/:name`.

Schemas

The second level in the three-level namespace hierarchy belongs to schemas. Using the list of all available catalogs, we can list all schemas present in each of the catalogs using the endpoint `api/2.0/unity-catalog/schemas`. The following code snippet shows the request pattern we used to test the endpoint. Alongside calling the endpoint a catalog name had to be passed as a parameter.

```
curl -n -v -X GET --header "Authorization: Bearer
    $DATABRICKS_TOKEN" -H 'Content-Type: application/json' https
    ://DATABRICKS_INSTANCE/api/2.0/unity-catalog/schemas -d '{"
    catalog_name": "NAME_OF_CATALOG"}'
```

We decided to skip the example response in this case since it was too long. We will show an example response with only the information necessary for the lineage computation in Section 4.3. In order to see the full list of fields that are returned please see the documentation [25].

To get information about only one schema the endpoint `<prefix>/schemas/:name` can be used.

Tables

Now that we know how to get all the catalogs and schemas, we can take a look at how to get to the most important data entities - tables. Tables are used to store data but are also a part of the Databricks lineage information. Unity Catalog API offers an endpoint for listing tables in specified catalog and schema.

The request is as follows:

```
curl -n -v -X GET --header "Authorization: Bearer
    $DATABRICKS_TOKEN" -H 'Content-Type: application/json' \
https://<INSTANCE_URL>/api/2.0/unity-catalog/tables -d '{"
    catalog_name": "CATALOG_NAME", "schema_name": "SCHEMA_NAME"}'
```

An example response alongside the useful information that can be used further in lineage processing can be found in 4.2. We decided not to include an example response here since it was too long. To see the full list of attributes that can be obtained, please see the documentation [26].

3.6.3 JDBC connection

As we mentioned before, the Unity Catalog API can be used only for instances that have Unity Catalog enabled. Since there still can be instances with Hive metastore only, there has to be a way of connecting to the metastore to obtain information. To do so, a JDBC connection to a cluster is used.

Let us now describe what is needed to successfully create the JDBC connection string in order to connect to the cluster. The string needs following parameters [7]:

```
jdbc:databricks://<Server Hostname>:443;HttpPath=<Http Path>[;
    property=value[;property=value]]
```

The only optional parameter is the `property` and all others are required. Hence users will have to provide information about (please see Figure 3.4 for visual representation):

- **Server Hostname** - the address of the server

– this can be found in the URL of the Databricks workspace or in the *Advanced options* section as the *Server Hostname*

- **Http Path** - the Databricks compute resources URL
  – this part can be found in the cluster details, under the *Advanced options* as the *HTTP Path*

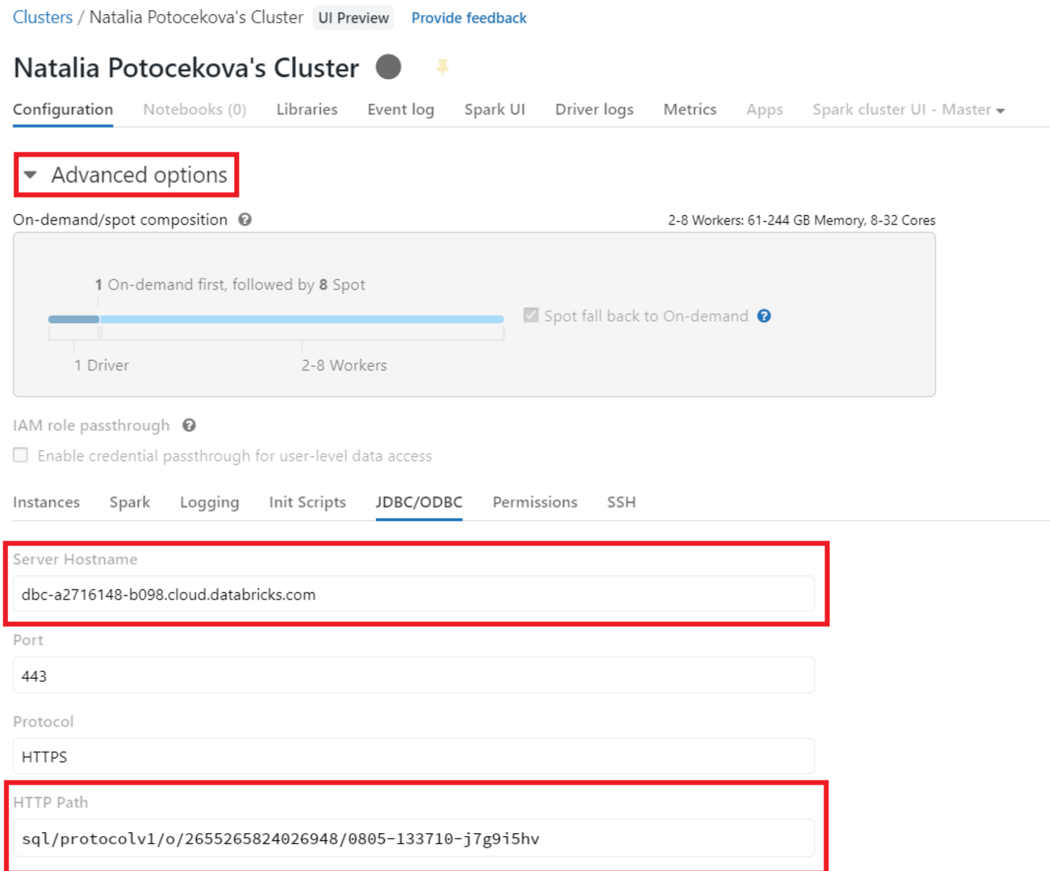


Figure 3.4: Cluster settings in Databricks

The following code snippet shows an example JDBC Databricks connection string created for testing purposes of the JDBC connector provided by Databricks.

```
"jdbc:databricks://dbc-a2716148-b098.cloud.databricks.com:443;
    HttpPath=sql/protocolv1/o/2655265824026948/0805-133710-
    j7g9i5hv";
```

Additionally to that, a user has to provide us the **Access token** with correct rights for working with clusters and reading data. This access token is then added to the properties under the name `PWD` which represents the password used for authentication.

4. Analysis

In this chapter, we discuss the concrete goals set for this thesis project, the technologies and tools that had to be explored, and the challenges that we faced from the data lineage point of view. Also, we do a thorough analysis of available data, tools, and concepts that are used in Databricks and are necessary in order to produce a lineage graph for the Databricks Platform.

The goal of this thesis is to write a tool that will be able to show lineage for Databricks notebooks. To be more precise there are several concrete steps that needed to be done:

1. Extraction of data from Unity Catalog and Hive metastore.

2. Using the extracted data to analyze the Databricks notebooks (Python and SQL cells).

3. Designing a solution that is easily extendable for other languages supported in Databricks (R, Scala).

4.1 Data necessary for lineage

In order to fulfill the set goals, we need to obtain the necessary data from Databricks first so that we could use them in the analysis later. In this section, we describe what kinds of data are necessary for data lineage display. Then in the following sections, we discuss how to obtain these data from different data sources that Databricks provide - Unity Catalog and Hive metastore.

There are two kinds of entities we could consider as important for data lineage:

1. The data entities (stored in possibly different formats)

2. The source codes that are working with the data and produce lineage

Let us examine what kind of data can be found in Databricks for each category.

Data

As we mentioned in Chapter 3 data in Databricks are stored as tables or views in different formats. Based on the object model described in Section 3.2.2, in order to get to the tables stored either in Unity Catalog or Hive metastore we need to know the available catalogs, schemas, and table or view names.

Additionally, Unity Catalog provides so-called lineage information. This lineage information is at most 30 days old data records that show which notebook worked with which tables. This information can be useful for us as a sort of backup mechanism. In case the analysis of a notebook using Manta scanners fails (for example some feature is not yet supported) the lineage information (if available) could be used to fill the gaps and show better lineage information in the final result.

The lineage information can be either table level or column level. The table-level lineage shows the direction of how data flows between tables in a given

notebook. An example of table-level lineage from Unity Catalog can be seen in Figure 4.1. The column-level lineage shows the exact columns from tables that participated in the data lineage flow. An example of a column-level lineage can be seen in Figure 4.2.

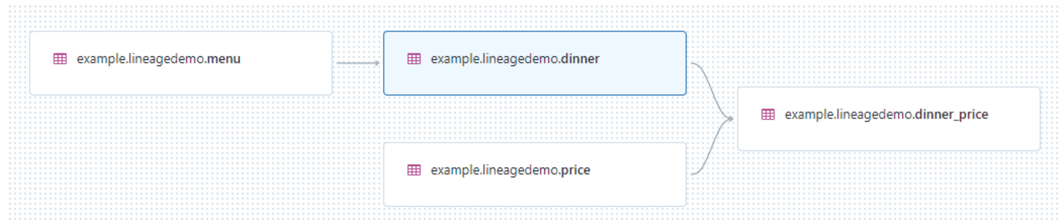


Figure 4.1: The example of table level lineage in Unity Catalog

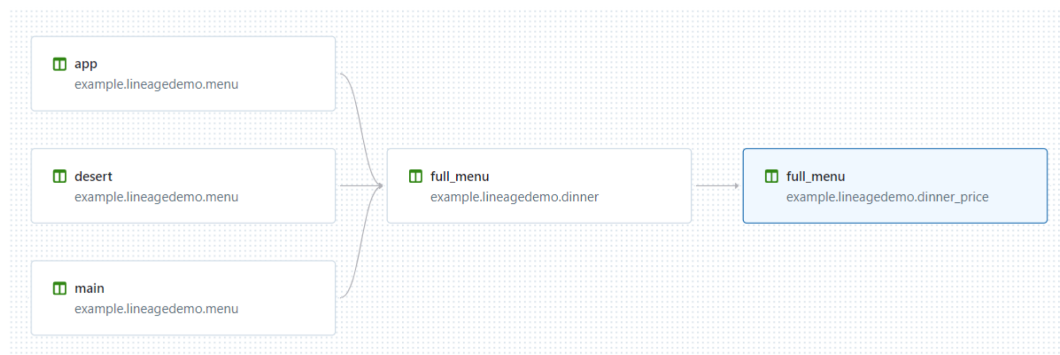


Figure 4.2: The example of column level lineage in Unity Catalog

Source codes

The second group of entities that need to be extracted are the source codes. As we mentioned in Section 3.4 there are two main source code types that are interesting for us - notebooks and queries. Exact details of how the Databricks notebooks look like can be found in Section 3.4. The queries are stored separately from the notebooks, hence we need to find another suitable export strategy for them as well.

Now that we know what kinds of data are necessary for data lineage, let us explore different options on how to obtain them. As we mentioned in Section 3.2.2 there are two main ways where data can be stored in Databricks - the Unity Catalog and Hive metastore. The following subsections discuss how we can obtain all data necessary for data lineage using tools that Databricks provides.

4.2 REST API

As we mentioned in the previous section there are two categories of data we need for data lineage - the data itself and programs that work with them. We already described interesting endpoints in Section 3.6.1. In this section, we discuss which methods are useful for computing lineage. In some cases, we show an example response, and what parts of it contain useful information. The unnecessary information in the response was replaced by three dots.

4.2.1 DBFS API 2.0

The first set of endpoints we mentioned in the previous chapter were endpoints for retrieving information about files stored in the Databricks file system (DBFS). For a single file or directory, we could obtain the following information:

```
{
  "path": "/tmp/HelloWorld.txt",
  "is_dir": false,
  "file_size": 13,
  "modification_time": 1622054945000
}
```

From the information available, the first three attributes could be useful. The path could be used as an identifier of the resource. Based on the information if the resource is dictionary or not we can make decisions about further processing and using the file size information we could for example filter those files that are too large or we can use this information to prepare a buffer large enough to store the file.

Another useful action is listing the contents of directories or getting more details about files. As we mentioned in Section 3.6.1 the **list** method returns a list of information for each file present in the directory. The information about a single file is the same as for a single file. Hence, the method could be useful in the manner that we would not have to make a separate request for each of the files present in the directory.

When obtaining the content of each file we can get the following information:

```
{
  "bytes_read": 8,
  "data": "ZWxsbywgV28="
}
```

This information can be used to properly store the files in our extraction folder.

4.2.2 Workspace API 2.0

Another set of endpoints we mentioned in Section 3.6.1 was related to notebooks and how to export them. In this section, we describe what has to be done to extract the notebooks properly using the REST API.

The first step is listing all available notebooks in a specified directory. For an easier description of the useful information, let us show the response for the **list** method again:

```
{
    "objects": [
        {
            "object_type": "NOTEBOOK",
            "path": "/Users/WORKSPACE_NAME/Data Lineage Test",
            "language": "PYTHON",
            "object_id": 754855114107838
        },
        {
            "object_type": "NOTEBOOK",
            "path": "/Users/WORKSPACE_NAME/UCtest",
            "language": "PYTHON",
```

```
13                  "object_id": 1582990291900299
14              },
15              {
16                  "object_type": "NOTEBOOK",
17                  "path": "/Users/WORKSPACE_NAME/anotherTest",
18                  "language": "PYTHON",
19                  "object_id": 1582990291900320
20              }
21          ]
22  }
```

Almost all of the fields we obtain in the response can be used further to obtain the notebook content. The `object_type` field defines if the item is a notebook or directory. In the case of a directory type, a recursive call would have to be made.

The `path` field can be used for exporting the concrete notebook using the export method described in the following section. And the `language` and `object_id` can be used for more detailed notebook description.

Using the information from the **list** method, we can call the **export** method to export the notebook in the specified format. These two steps are all that needs to be done in order to export notebooks from Databricks using the REST API.

4.2.3 Jobs API 2.1

As we mentioned in Section 3.6.1 jobs can be used to execute notebooks automatically based on some parameters. Hence, we would like to know how to extract them, in case we would like to visualize them in the final graph as well.

In order to do so, the **list** method in the Jobs API would have to be called. The following code sample contains an example response mentioned in the documentation [21]. Since the response was too long, we only show those parts of the response that we considered useful.

```
1  [
2    {
3    "jobs": [
4      {
5          "job_id": 11223344,
6          ...
7            "tasks": [
8              {
9                "task_key": "Sessionize",
10               "description": "Extracts session data from events",
11               "depends_on": [],
12               "existing_cluster_id": "0923-164208-meows279",
13               "spark_jar_task": {
14                 "main_class_name": "com.databricks.Sessionize",
15                 "parameters": [
16                   "--data",
17                   "dbfs:/path/to/data.json"
18                 ]
19               },
20               "libraries": [
21                 {
22                   "jar": "dbfs:/mnt/databricks/Sessionize.jar"
23                 }
24               ],
```

```
          ...
        },
        {
          ...
        },
        {
          "task_key": "Match",
          "description": "Matches orders with user sessions",
          "depends_on": [
            {
              "task_key": "Orders_Ingest"
            },
            {
              "task_key": "Sessionize"
            }
          ],
          ...,
          "notebook_task": {
            "notebook_path": "/Users/user.name@databricks.com/
  Match",
            "base_parameters": {
              "name": "John Doe",
              "age": "35"
            }
          },
          ...
        }
      ],
      "...
}
]
```

The most useful information we can obtain calling this endpoint is the ID of the job, the tasks it executes, their names, what they depend on, what data and libraries they use, what notebooks they execute and what are their default parameters. All of this information can be used in case we would like to visualize the jobs in the result graph.

4.2.4 Queries API

Last but not least source code entities we would like to extract are queries. Queries are one-cell executables that are written in SQL and can manipulate with data entities.

To get a list of all available queries the **list** method from the Queries API group has to be called. Note that due to the fact that the response was really long, we show only the parts important for the lineage. The missing parts were replaced by three dots. The complete full response can be found in the Documentation [15].

```
{
  ...
  "results": [
    {
      "query_hash": "string",
      "parent": "string",
      "name": "string",
      "permission_tier": "CAN_VIEW",
```

```
      "description": "string",
      "tags": [
        "string"
      ],
      ...
      "query": "string",
      "data_source_id": "string",
      "user_id": 0,
      ...
    }
  ]
}
```

The most important parts of the response are the fields `query` which contains the SQL code that represents the query and the `name` field which contains the name of the query. A possible useful field in the future could be the `data_source_id` which identifies the concrete warehouse where the query is stored. Should we ever want to visualize this information, this field would be necessary to do so.

4.2.5 Data sources

In case we would like to visualize different warehouses users have in their instance, we would have to use the data sources endpoint described in Section 3.6.1. The response obtained from such a request would look as follows:

```
[
  {
    "id": "f7df1dfd-565d-4506-accb-8a1e0f8fad09",
    "name": "starter-warehouse",
    "pause_reason": null,
    "paused": 0,
    "supports_auto_limit": true,
    "syntax": "sql",
    "type": "databricks_internal",
    "view_only": false,
    "warehouse_id": "3d939b0cc668be06"
  }
]
```

When visualizing the warehouses we could use the `id` and `name` fields for the warehouse identification.

4.2.6 Authorization

In order to be able to use the REST API one needs to be authenticated first. For this purpose *personal access token* (PAT for short) or *passwords* can be used. Tokens are preferred for authentication and are automatically enabled for all Databricks accounts.

In order to work with the REST API we would need PAT from our customers directly since there is no other way of getting to them. For example, they could provide it in some sort of configuration. On the other hand, there is a possible issue that customers will not be willing to share the tokens due to security reasons (they could be worried that the token will be stored somewhere or that it will

grant access to the whole API which means also write operations). This could however be solved by using tokens with limited access rights.

4.3 Unity Catalog API

As we mentioned in Section 3.6.2 Databricks created an extension of the REST API called Unity Catalog API that provides information about data entities stored in metastores and also lineage information related to them. In the following sections, we focus on important methods in order to compute the lineage properly.

For all following endpoints the common prefix is: `/api/2.0/unity-catalog`

Metastores and External Locations

The top-level part of the whole Databricks architecture is the metastore. Metastore contains all the information about the data entities stored inside of it as well as the data entities themselves. In case users use multiple metastores in their instance, it could be useful to show them in the final graph. Hence we need to be able to obtain the metastore information. As we described in 3.6.2 there are methods in the Unity Catalog API to do so.

The following response was returned when calling the `<prefix>/metastores` endpoint which returns information about all metastores used by the instance. Note that since the response was too long we only show the parts of the response we considered useful for further use. Other parts were replaced by the three-dot notation. To see the full example response please see the documentation [24].

```
{
    "metastores": [
        {
            "name": "databricksshared_metastore",
            "storage_root": "s3://databricksshared/HERE-WAS-ID-OF-METASTORE",
            ...,
            "metastore_id": "HERE-WAS-ID-OF-METASTORE",
            ...,
            "cloud": "aws",
            "region": "us-east-1",
            "global_metastore_id": "aws:us-east-1:HERE-WAS-ID-OF-METASTORE"
        },
        {
            "name": "myunitycatalogmetastore",
            "storage_root": "s3://myunitycatalogs3bucket/HERE-WAS-ID-OF-METASTORE",
            ...,
            "metastore_id": "HERE-WAS-ID-OF-METASTORE",
            ...,
            "cloud": "aws",
            "region": "us-east-1",
            "global_metastore_id": "aws:us-east-1:HERE-WAS-ID-OF-METASTORE"
        }
    ]
}
```

The most important metastore information is the name and the storage root path. Also, the global metastore ID is really important in case users want to scan more than one Databricks instance.

As we also mentioned in Section 3.6.2 we can obtain information about External Locations as well. This information could be useful in case we wanted to connect the Databricks results with the results of other scanners in Manta.

Catalogs

When it comes to data entities the topmost entity is the catalog. As mentioned in Section 3.6.2 there are methods available that gather information about the catalogs.

An example response we can get for catalogs is shown as a code snippet below. Please note, that since the response was long, we show only the useful information, and everything else has been replaced with a three-dot notation. In order to see the full list of fields that are returned please see the documentation [23].

```
{
    "catalogs": [
        {
            "name": "example",
            ...
            "metastore_id": "METASTORE_ID",
            ...
            "catalog_type": "MANAGED_CATALOG"
        },
        {
            "name": "system",
            ...
            "metastore_id": "METASOTRE_ID",
            ...
            "catalog_type": "SYSTEM_CATALOG"
        }
    ]
}
```

From this information, the most important one is the name of the catalog and possibly catalog type and metastore ID.

Schemas

On the second level of the namespace, there are schemas. Similarly to catalogs, schemas have their API endpoints as well.

For an example request, we obtained a response shown as a code snippet below. Please note, that since the response was long, we show only the useful information, and everything else has been replaced with a three-dot notation. In order to see the full list of fields that are returned please see the documentation [25].

```
{
    "schemas": [
        {
            "name": "default",
            "catalog_name": "complicatedlineage",
            ...
            "metastore_id": "METASTORE_ID",
```

```
                "full_name": "complicatedlineage.default",
                ...
            },
            {
                "name": "information_schema",
                "catalog_name": "complicatedlineage",
                ...
                "metastore_id": "METASTORE_ID",
                "full_name": "complicatedlineage.information_schema",
                ...
            },
            {
                "name": "lineagedemo",
                "catalog_name": "complicatedlineage",
                ...
                "metastore_id": "METASTORE_ID",
                "full_name": "complicatedlineage.lineagedemo",
              ...
            }
        ]
}
```

As can be seen in the example result above, there is always the default schema created by the system, custom schemas created by users and then there is an information schema created by the system as well.

Tables

Tables alongside views reside on the third level in the namespace. Unity Catalog API provides methods for obtaining information about tables as we mentioned in Section 3.6.2.

In order to get full lineage we need to iterate over all tables and their columns and request lineage for them separately. Then the full lineage can be computed by connecting the separate lineages based on common nodes.

To be able to iterate over the list of tables, we need to get it first. Unity Catalog API offers an endpoint for listing tables in specified catalog and schema.

Let us describe the response on an existing example.

Suppose we have the following structure:

- catalog name: `complicatedlineage`
- schema name: `lineagedemo`
- we have 4 tables in the schema named `dinner`, `dinner_price`, `menu`, `price`

The following code snippet shows a response to such a request. Note that since the response was too long we show only the information useful for further processing and the unnecessary information was replaced by the three-dot notation. To see the full list of attributes that can be obtained, please see the documentation [26].

```
{
    "tables": [
        {
            "name": "dinner",
            "catalog_name": "complicatedlineage",
            "schema_name": "lineagedemo",
```

```
            "table_type": "MANAGED",
            "data_source_format": "DELTA",
            "columns": [
                {
                    "name": "recipe_id",
                    ...
                    "type_name": "INT",
                    ...
                },
                {
                    "name": "full_menu",
                    ...
                    "type_name": "STRING",
                    ...
                }
            ],
            "storage_location": "s3://LOCATION",
            ...,
            "metastore_id": "METASTORE_ID",
            "full_name": "complicatedlineage.lineagedemo.dinner",
            ...
        },
        {
            "name": "dinner_price",
            "catalog_name": "complicatedlineage",
            "schema_name": "lineagedemo",
            "table_type": "MANAGED",
            "data_source_format": "DELTA",
            "columns": [
                {
                    "name": "recipe_id",
                    ...
                    "type_name": "INT",
                    ...
                },
                {
                    "name": "full_menu",
                    ...
                    "type_name": "STRING",
                    ...
                },
                {
                    "name": "price",
                    ...
                    "type_name": "DOUBLE",
                    ...
                }
            ],
            "storage_location": "s3://LOCATION",
            ...
            "metastore_id": "METASTORE_ID",
            "full_name": "complicatedlineage.lineagedemo.
    dinner_price",
            ...
        },
        {
            "name": "menu",
            "catalog_name": "complicatedlineage",
```

```
            "schema_name": "lineagedemo",
            "table_type": "MANAGED",
            "data_source_format": "DELTA",
            "columns": [
                {
                    "name": "recipe_id",
                    ...
                    "type_name": "INT",
                    ...
                },
                {
                    "name": "app",
                    ...
                    "type_name": "STRING",
                    ...
                },
                {
                    "name": "main",
                    ...
                    "type_name": "STRING",
                    ...
                },
                {
                    "name": "desert",
                    ...
                    "type_name": "STRING",
                    ...
                }
            ],
            "storage_location": "s3://LOCATION",
            ...
            "metastore_id": "METASTORE_ID",
            "full_name": "complicatedlineage.lineagedemo.menu",
            ...
        },
        {
            "name": "price",
            "catalog_name": "complicatedlineage",
            "schema_name": "lineagedemo",
            "table_type": "MANAGED",
            "data_source_format": "DELTA",
            "columns": [
                {
                    "name": "recipe_id",
                    ...
                    "type_name": "LONG",
                   ...
                },
                {
                    "name": "price",
                    ...
                    "type_name": "DOUBLE",
                    ...
                }
            ],
            "storage_location": "s3://LOCATION",
            ...
            "metastore_id": "METASTORE_ID",
```

```
                "full_name": "complicatedlineage.lineagedemo.price",
                ...
            }
        ]
}
```

The only useful information for this matter is the table names, column names, and possibly the column type, full table name, location, and metastore ID.

4.4 Prototype

In order to test if the Unity Catalog API could be used to recreate Databricks lineage in the Manta platform, we have decided to create a simple prototype script that would extract lineage information for a specified catalog and schema. This prototype has been developed and tested in cooperation with another member of the Manta team.

The prototype works as follows:

1. Python script downloads the lineage information for all tables and columns in the specified catalog and schema.
    - The lineage has to be downloaded for each table and column separately since Databricks provides only the lineage information that is directly related to the table.
2. The results are saved into PostgreSQL and using transformations are converted into CSV files that can be used in Manta.
    - The JSON results were manually inserted into the database by the Manta employee that used our Python script mentioned above for the extraction.
3. Using the CSV files Manta can create a graph for the lineage.

The example result of this prototype was the graph shown in Figure 4.3.

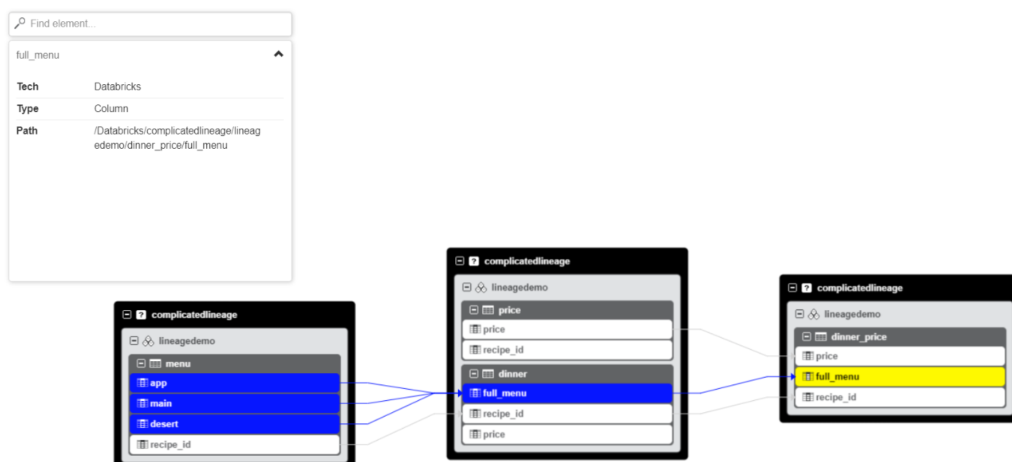


Figure 4.3: Example graph from prototype

When we compare it to the graph from Databricks UI we can see that it is exactly the same lineage, but the graphs for the tables and columns are merged into one. (See Figure 4.1 and Figure 4.2 for Databricks lineage information)

4.5 Hive metastore

The original way of storing data in Databricks used to be the Hive metastore. Even though Databricks introduced the Unity Catalog many users keep their Hive instances untouched and have not yet migrated to the Unity Catalog. Due to this we also have to analyze how to retrieve the data information from the Hive metastore.

Since Databricks provides REST API for exporting notebooks and source codes, we only need to analyze how to obtain information about tables and views and what kind of information we are able to obtain. The following sub-sections will discuss possible options for data retrieval from the Hive metastore.

4.5.1 Unity Catalog API behavior

We tried calling Unity Catalog API on the Databricks instance that does not have Unity Catalog enabled. We tried the API call for listing all tables and getting lineage information for a table. The following code samples will show both requests and the results we obtained.

For listing all tables we used the following request:

```
1 curl -X GET --header "Authorization: Bearer $DATABRICKS_TOKEN"
2 -H 'Content-Type: application/json'
3 https://<INSTANCE_URL>/api/2.0/unity-catalog/tables
```

The obtained response was as follows:


```
1  {
2      "error_code": "METASTORE_DOES_NOT_EXIST",
3      "message": "No metastore assigned for the current workspace."
   ,
4      "details": [
5          {
6              "@type": "type.googleapis.com/google.rpc.RequestInfo"
   ,
7              "request_id": "24381936-1ed5-4153-ae21-221e1e1d6bfe",
8              "serving_data": ""
9          }
10     ]
11 }
```


As can be seen in the response, we obtained an error response with a message that there is no Unity Catalog workspace assigned for the instance. Hence we cannot use the Unity Catalog to obtain any information about tables stored in the Hive metastore.

The second thing that we needed to verify was if there were any lineage information kept for Hive metastore tables. For that purpose, we tried the following command for obtaining lineage information for the table `example.lineagedemo.dinner`.

```
curl -n -v -X GET --header "Authorization: Bearer
    $DATABRICKS_TOKEN"
-H 'Content-Type: application/json'
https://dbc-6236ff65-3cdb.cloud.databricks.com/api/2.0/lineage-
    tracking/table-lineage
-d '{"table_name": "example.lineagedemo.dinner", "
    include_entity_lineage": true}}'
```

We obtained the following response:

```
{
    "error_code": "PERMISSION_DENIED",
    "message": "Cannot resolve metastore id from UserContext(
    workspaceId: 2640600745528909)."
}
```

As can be seen in the response, this API method cannot be used for obtaining lineage information for the tables.

4.5.2 Hive metastore access

The original plan for accessing the Hive metastore in Databricks was to use the Hive scanner from Manta. Firstly the analysis using the Hive scanner would be done and then the following analysis would build on top of the scanner results. However, we could not proceed with this plan as the Hive metastore support has been disabled in the R38 version of Manta. Hence we needed to explore new options how to obtain information from the metastore.

JDBC spark connection

In the cluster settings in Databricks, there is a JDBC spark connection available. We decided to explore options that could use this connection and extract the information from the metastore. We tried the following options:

1. JayDeBe in python [20]
2. simba in spring [18]
3. JDBC connector in Java

However, using the spark JDBC drivers was not successful. Either the proper driver could not be found or there were issues when trying to access the metastore.

The second option we found was to create a `jdbc:databricks` connection string and use Databricks custom JDBC connector. This solution was the only one that was successful when connecting to the Databricks metastore. The following section describes this solution in more detail.

JDBC connection string

For the internal hive metastore, the only option that seemed to be working was using the official JDBC connector [12] and connecting to the clusters in order to obtain any information about the metastore content. For a brief overview of what can be seen when connecting to the cluster, we have used the DBeaver

tool which has a Databricks plugin and shows the content in a more understandable way (note that the DBeaver has an official plugin from Databricks and is hence equipped to show the Unity Catalog content as well as the Hive metastore content).

The main difference between the Hive metastore and Unity Catalog is that Unity Catalog uses a three-level namespace and Hive metastore uses only two levels. Using DBeaver for brief data exploration gave us additional important information. For instances where both Hive metastore and Unity Catalog metastores are present, the full names of the tables need to be normalized to a three-level namespace format. That is done in a simple manner. Databricks creates an abstraction as if the Hive metastore contained one catalog called `hive_metastore`. Hence everything that is located in the Hive metastore (schemas and tables/views) is attached under this catalog. Figure 4.4 shows a screenshot of the DBeaver displaying the content of Unity Catalog (green rectangle) and Hive metastore (red rectangle) parts.

Figure 4.4: The DBeaver displaying Unity Catalog and Hive metastore contents

We described how to create a proper JDBC connection string in Section 3.6.1. Now that we know how we can successfully connect to the Databricks cluster using the JDBC connection string the only thing left is to figure out how to get the table and schema-level information to our scanner. In order to do so, we need to use SQL queries. To get the available schemas in the Hive metastore we need to execute the `SHOW DATABASES` query. This will return a list of schema names that are present in the Hive metastore and are accessible using the connection string provided by the user (we need to take into account that users may not have access to all tables and schemas). When we obtained the list of schemas, we need to get the list of tables and views for all of them. This can be done using two queries. Firstly we need to use the `USE schemaName` command to set the currently used schema to the one we want to query. Then the `SHOW TABLES` query needs to be called to obtain the list of all tables present in the schema (again only those that

are accessible with respect to the user rights and privileges). When listing views, the first step is the same and the second uses the `SHOW VIEWS` query instead. In order to get the names of tables or view columns the `SHOW COLUMNS FROM tableName` query needs to be used.

The information about tables that can be obtained from the Hive metastore is far more limited than the information provided by the Unity Catalog API described in Section 4.3. The information available from Hive metastore is:

- table name
- name of each column of the table
- a Boolean value indicating if a table is temporary

4.6 Databricks notebooks

Previous sections focused on data entity retrieval and extraction. Now that we know possible ways how to do so, let us focus on the source codes that work with the data entities. In this section, we describe Databricks notebooks in more detail.

Notebook in Databricks is similar to Jupyter Notebook used commonly in Python. It is a collection of cells that can contain either some text information or scripts written in supported language that can do some sort of transformation with data. These computations are usually run on an Apache Spark cluster. In Databricks, notebooks are created in workspaces.

4.6.1 Export

Databricks scanner has to call other Manta scanners like for example Python scanner for analysis of the Python cells. Due to this, the notebooks have to be exported in some way so that scanners could work with them. As we discussed in Section 4.2.2 to export the notebooks we can use the REST API provided by Databricks. However, there are multiple options for the export format as we mentioned in Section 3.4. Let us take a deeper look at each of them and discuss which option is the most suitable for the Databricks scanner.

DBC Archive

The first option for a single notebook export is a DBC archive which is a binary format. To use this format we would have to know how to properly extract it and how to work with the source codes that are stored in it. We tried to work with the DBC archive ourselves and we found out that it behaves similarly to a standard ZIP archive. Hence extraction would not be a problem. However, when we successfully extracted the archive we found out, that the source code format is not suitable at all. The source code is stored in a JSON structure that contains a lot of metadata information alongside the executable notebook source codes. This format would be difficult to work with and lots of pre-processing would be needed to put it in a somewhat usable form.

Source File

The second option is to extract the notebook as a source file. That means that the notebook is downloaded as `.scala`, `.py`, `.sql`, or `.r` file based on the default notebook language. An example of this format can be found in Section 3.4.

To use this file format, some pre-processing needs to be done before the file is passed to the language scanner. The pre-processing has to include deleting the comments before the source code and filtering only cells that were written in a given language.

Even though some pre-processing has to be done, the required changes are not as extensive as they would be in the case of the DBC archive.

IPython Notebook

Another option for notebooks export is the IPython notebook file. It is a Jupyter notebook with the extension `.ipynb`. This file format could be useful in the future when the Python scanner would support Jupyter Notebooks. However, as of now, the Python scanner cannot work with the Jupyter Notebooks files and since this format contains a lot of javascript and metadata, it would be harder to prepare it to a suitable form similarly as we mentioned in the case of DBC archives.

HTML

The last option for notebooks export is the HTML file. We do not find this format suitable for scanner usage because it uses a lot of javascript, the code is not easy to read nor parse and it would be hard to write some sort of reasonable processing for such files.

To summarize, we decided that the best option for exporting notebooks would be the source file option since it requires the least amount of pre-processing, and the structure and contents of the format are determined (since this format does not contain any metadata that would make processing harder).

4.7 Spark context

As we know from Section 3.4.2, Databricks notebooks support the language interaction using the spark context. In the following sections, we describe what kinds of information can be passed from one language to another and also what kinds of operations can be done with them.

Scala

The first language we take a look at is Scala. In Scala both read and write operations can be done to spark session. For both actions, there are special methods that are used in the source code. For reading it is the `spark.conf.get` method and for writing it is the `spark.conf.set` method. The following code snippet shows how to use these methods.

```
%scala
// Setter:
spark.conf.set("var.scala_var", "my_value_scala")
// Getter:
val var_scala = spark.conf.get("var.scala_var")
```

Python

Just like in Scala, Python can perform both read and write operations. In order to do so, the methods `spark.conf.get` and `spark.conf.set` can be used. The following code shows an example usage of these methods.

```
# Setter:
spark.conf.set("var.python_var", "my_value_python")
# Getter:
var = spark.conf.get("var.python_var")
```

R

Similarly to Python and Scala, R language can too perform both read and write operations. However, the syntax is more complicated than in the first two languages. The following code snippet shows both setting a variable value and getting a variable value using spark methods.

```
# Setter:
setEnvVar <- function(var_name, var_value) {
  list_param <- list()
  list_param[[var_name]] <- as.character(var_value)
  SparkR::sparkR.session(sparkConfig = list_param)
  return(TRUE) # to avoid return a session
}
setEnvVar("var.r_var", "my_value_r")

# Getter:
getEnvVar <- function(var_name) {
  return(unname(unlist(SparkR::sparkR.conf(var_name))))
}
var <- getEnvVar("var.r_var")
```

SQL

The last supported cell language is SQL. In SQL reading the value of a variable can be done using a simple construct `${variable_name}`. The following code snippet shows the usage of this concept.

```
select * from ${var.table_name}
```

Hence we see that the SQL language can perform the read operation. However, what about writing?

As of now, it seems that the SQL language can only read the variable value. However, according to the article about SQL results [14], there is an option to investigate the latest SQL query result directly from the python cell through the `_sqldf` variable (dataframe). This feature was supposed to be already released but due to some issues it has been delayed and will be available in the Databricks

3.74 through 3.76 [4].

There is also an option to share dataframes (table-like structures) through the spark context as well. In order to do so a spark dataframe has to be created and saved to the spark context [17].

Let us now describe how this can be done in different languages.

Scala

Again the first language we look at is Scala. In order to pass a dataframe to a spark context a method `createOrReplaceTempView` has to be called on an existing dataframe. This method then creates a temporary view with a specified name in the spark context. The temporary view can be then accessed as a normal table or view, however, only by the following cells of the given notebook. An example below shows how to use the mentioned method.

```
%scala
val nb = Seq(1, 2, 3).toDF("nb") //Creates a spark dataframe "nb"
nb.createOrReplaceTempView("nb_tmp") //Creates a temp table "
    nb_tmp"
```

The `nb_tmp` can now be used in any language for example in the SQL to do the following query:

```
%sql
SELECT * FROM nb_tmp
```

Python

The concept of creating a temporary table in spark context stored in the spark session is the same for Python and Scala.

Again the method `createOrReplaceTempView` needs to be called on an existing dataframe. The following example shows how this can be done in Python.

```
%python
dept = [("Finance",10),
        ("Marketing",20),
        ("Sales",30),
        ("IT",40)
      ]
deptColumns = ["dept_name","dept_id"]
deptDF = spark.createDataFrame(data=dept, schema = deptColumns)
deptDF.createOrReplaceTempView("deptDF_tmp")
```

The `deptDF_tmp` can now be used in any language for example in the SQL to do the following query:

```
%sql
SELECT * FROM deptDF_tmp
```

R

In order to pass dataframes from R to other languages few steps need to be followed. The first step is to install all packages that contain methods for working with spark. In this case, the packages needed are the `magrittr` and `dplyr`. Both

of the packages then need to be included as libraries in the code. Also in order to be able to work with Spark a whole backend need to be included using the `require(SparkR)` call. After all of these steps are fulfilled, the dataframe can be stored using the `createOrReplaceTempView` method. The code snippet below shows all steps that need to be done.

```
%r
install.packages("magrittr") # package installations are only
    needed the first time you use it
install.packages("dplyr")    # alternative installation of the
    %>%
library(magrittr) # needs to be run every time you start R and
    want to use %>%
library(dplyr)
require(SparkR)
# Create list with 5 elements
my_input_list = list(56,78,90,45,67)

# Convert dataframe to list using data.frame()
df <- data.frame(my_input_list) %>%
            SparkR::as.DataFrame()
df

createOrReplaceTempView(df, "test_r")
```

SQL

As was mentioned in the previous examples, the SQL language can read the dataframes created in Scala, Python, or R as if they were normal tables or temporary views. The write operation is not possible yet, the only write-like operation is saving the latest SQL query result to the `_sql` variable which we mentioned in the previous sections discussing the Spark context and passing variables.

4.8 External sources

One of the many advantages of the Manta Flow platform is that it can show data lineage between different systems. For example, if some BI tool uses a database as a data source Manta can show the connected lineage when users scan both technologies. As we learned in Section 3.5 Databricks supports integration with several external sources. In the following sections, we focus on the technologies that are important for Manta and can be used in Databricks.

Source systems

In this section, we describe how data can be loaded from some external database (source system) into Databricks.

Let us now show how data can be loaded from Oracle. For other source systems, such as Teradata or DB2 the data can be obtained in the same way as for Oracle.

In order to load data from Oracle to Databricks, JDBC is used. To do so, the correct driver has to be installed on the cluster.

Setting the connection to Oracle can be done in the following way:

```
driver = "cdata.jdbc.oracleoci.OracleOCIDriver"
url = "jdbc:oracleoci:RTK=5246...;User=myuser;Password=mypassword
    ;Server=localhost;Port=1521;"
```

Then when everything is set up properly, data can be loaded like this:

```
remote_table = spark.read.format ( "jdbc" )
                         .option ( "driver" , driver)
                         .option ( "url" , url)
                         .option ( "dbtable" , "Customers")
                         .load ()
```

Data from Oracle are available only in the notebook that loaded them. If other notebooks want to use these data as well, they first need to be saved as a table in Databricks or loaded by the notebook itself from Oracle. The following snippet shows how the data can be saved as a table.

```
remote_table.write.format("parquet").saveAsTable("SAMPLE_TABLE")
```

Consumers

The Consumer technology we focus on is PostgreSQL. Writing data into SQL databases is also done through the JDBC connection. The spark write function is called in order to do so. An example of how Dataframe can be saved into PostgreSQL using spark can be found below:

```
studentDf.select("id","name","marks").write.format("jdbc")\
    .option("url", "jdbc:postgresql://localhost:5432/dezyre_new")
    \
    .option("driver", "org.postgresql.Driver").option("dbtable",
   "students") \
    .option("user", "hduser").option("password", "bigdata").save
   ()
```

4.9 Databricks SQL vs Hive SQL

Since Databricks notebooks can contain cells written in SQL we needed to find a way to analyze the SQL code in Manta. The initial idea was to use the existing Hive scanner in order to do so since Databricks uses Hive metastore as one of the possible options for storing metadata. However, as we found out, Databricks created their own version of the SQL called Databricks SQL which is used in the notebook cells. In order to determine if we still could use the Hive scanner or if we needed a new one specialized in Databricks SQL we took a look at the concepts Databricks SQL provides and compared it with Hive. The following sections focus on the differences between the Hive and Databricks SQL dialects when it comes to concepts that have a real impact on displayed lineage.

4.9.1 Queries not parsed by Hive

When we were trying to figure out the differences between Hive and Databricks SQL dialects we found several queries and concepts that were not supported by Hive. The initial issue is that Hive does not support a three-level namespace. Hence no table names could be parsed properly when written in their full name

(e.g.: `c1.s1.t1`). Another issue is that there are some syntactic differences between Hive and Databricks SQL and queries that are possible in one but not possible in the other. The following subsections contain those that could in our opinion have an impact on lineage.

Catalogs

The first problem we encountered was that any query with keyword `CATALOG` in it would not be parsed since Hive does not support the three-level namespace. Hence if a user creates a new catalog using query `CREATE CATALOG catalogName;` Hive scanner would not be able to parse it and we would not have information that a new entity has been created. This is a sort of action that impacts lineage greatly and hence we would like to see it in a graph. The same would be true for dropping (deleting) catalogs using `DROP CATALOG catalogName CASCADE;`

External location

Another issue is that Hive does not support external locations at all. Hence creating or updating an external location would not be parsed and hence would not be present in the result graph. The following code samples contain the queries for creating and altering the external locations:

```
1 CREATE EXTERNAL LOCATION s3_remote URL 's3://us-east-1/location'
2      WITH (STORAGE CREDENTIAL s3_remote_cred)
3      COMMENT 'Default source for AWS exernal data';
```

```
1 ALTER EXTERNAL LOCATION descend_loc RENAME TO decent_loc;
```

Another example where external locations are important from a lineage point of view is when users create tables and store them in an external location. The following query shows the table creation in an external location.

```
1 CREATE TABLE main.default.sec_filings LOCATION 's3://depts/
    finance/sec_filings';
```

Since we would like to see the external locations in the result graph because it connects Databricks and other technologies together, this is a very important feature that is not supported by Hive.

Create tables and views

In Databricks SQL tables can be created using different methods and source formats. For example, the following query shows how to create a table using a CSV file which is not supported in Hive SQL.

```
1 CREATE TABLE student USING CSV LOCATION '/mnt/csv_files';
```

Another feature related to table creation is creating tables with generated columns. This can be done using the `GENERATED` keyword as shown in the following example.

```
1  CREATE TABLE rectangles(a INT, b INT, area INT GENERATED ALWAYS
    AS (a * b));
```

Neither of these features is supported by Hive.

Databricks SQL is also less strict on the keyword order. For example, the following query that specifies table comments and properties will not pass in Hive SQL. However, if we swapped the `TBLPROPERTIES` and `COMMENT` keywords, the command would pass.

```
CREATE TABLE student (id INT, name STRING, age INT)
    TBLPROPERTIES ('foo'='bar')
    COMMENT 'this is a comment';
```

Databricks SQL is also less strict on how the values need to be passed. The following example creates a table with user-defined table properties without passing values as a string.

```
CREATE TABLE T(c1 INT) TBLPROPERTIES('this.is.my.key' = 12, this.
   is.my.key2 = true);
```

This cannot be parsed by Hive SQL. However, when we change the query to pass strings (as shown in the following example) the query can be parsed.

```
CREATE TABLE T(c1 INT) TBLPROPERTIES('this.is.my.key' = '12', '
   this.is.my.key2' = 'true');
```

Regarding views, Databricks SQL provides an option to create a temporary view. This construct is not supported in Hive SQL at all. Since this concept is widely used by Databricks users we would lose an important part of the lineage in the result graph should there be no support for this feature. The following code shows an example of creating a temporary view.

```
CREATE TEMPORARY VIEW subscribed_movies
    AS SELECT mo.member_id, mb.full_name, mo.movie_title
         FROM movies AS mo
         INNER JOIN members AS mb
            ON mo.member_id = mb.id;
```

Another feature related to views is supporting multiple Lateral views in one statement. For example, the following query would not pass.

```
SELECT * FROM person
    LATERAL VIEW EXPLODE(ARRAY(30, 60)) tableName AS c_age
    LATERAL VIEW EXPLODE(ARRAY(40, 80)) AS d_age;
```

However, when using only one lateral view the query passes with no issues.

```
SELECT * FROM person
    LATERAL VIEW EXPLODE(ARRAY(30, 60)) tableName AS c_age
```

Alter tables

When working with tables we ran into an issue with altering tables. To be more precise we had issues with queries that tried to alter the table by using foreign keys. The following code samples contain an example problematic query:

```
ALTER TABLE pets ADD CONSTRAINT pets_persons_fk FOREIGN KEY(
   owner_first_name, owner_last_name) REFERENCES persons;
```

According to this article [16] Hive does not support foreign keys. Another problem in our Hive parser could be missing columns. Either way, we would not have information about changing a table if did not have a parser that could analyze such a code.

Another issue was with renaming columns. When we used the query `ALTER TABLE StudentInfo RENAME COLUMN name TO FirstName;` the parser had issues with the `RENAME COLUMN` keywords. Again this is a very important feature since we would like to show users when some column has been renamed.

Lambda functions

A new feature that is not supported in Hive at all is the lambda function which is a parametrized expression that can be passed to a function. Using this lambda function users can control the behavior of the expression. The following example shows a lambda function for sorting.

```
(p1, p2) -> CASE WHEN p1 = p2 THEN 0
                 WHEN reverse(p1) < reverse(p2) THEN -1
                 ELSE 1 END
```

The following example will show how the lambda functions can be used in the SELECT statements.

```
SELECT array_sort(array('Hello', 'World'),
  (p1, p2) -> CASE WHEN p1 = p2 THEN 0
              WHEN reverse(p1) < reverse(p2) THEN -1
              ELSE 1 END);
```

In order to properly show lineage for the latest example we would need to have a scanner that would be able to parse and analyze these functions. Hence this is another important feature that is not supported in Hive at all.

USE commands

The `USE SCHEMA schemaName;` command is used to set the current default schema to the one specified in the command. We need to be able to detect this as it dictates which tables are being used when no schema is specified. Since Hive supports only two-level namespace, the command that is used instead of this is `USE default;`. Another command that is related to this issue and is not supported by Hive is the `USE CATALOG catalogName;` command. This command sets the default catalog which is used when no catalog is specified in a table name. Both of these features are extremely important in order to properly show the lineage.

FROM VALUES statements

When exploring queries that specify some values using the `FROM VALUES` statements we found out that these statements are not supported in Hive. This can be problematic in situations when we use these statements in queries that produce lineage like `SELECT` queries for example `SELECT c1 FROM VALUES(1) AS T(c1);`. Another issue related to the `FROM VALUES` statements was the field name queries. For example, following query could not be parsed by the Hive scanner:

```
SELECT addr.address.name
    FROM VALUES (named_struct('address', named_struct('number',
   5, 'name', 'Main St'),'city', 'Springfield')) as t(addr);
```

Another example is in a selection query that uses expressions.

```
SELECT c1 + c2 FROM VALUES(1, 2) AS t(c1, c2);
```

Since the Hive scanner would not be able to parse these queries we would not see any lineage information in the result graph regarding these examples and hence would lose an important part of the lineage information.

Functions

The functions in Hive and functions in Databricks have different ways to be defined. In Hive it is defined in a programming language and then referenced by name like this:

```
CREATE TEMPORARY FUNCTION country AS 'com.hiveudf.employeereview.
    Country';
```

On the other hand, Databricks SQL has a special syntax for function definition. An example of a function definition in Databricks SQL is in a code sample below.

```
CREATE FUNCTION area(x INT, y INT) RETURNS INT
RETURN area.x + y;
```

Since functions can be used in queries that are important for lineage, this feature is important in order to gain complete lineage knowledge.

JSON Path expressions

Hive SQL does not support JSON paths at all. Hence when used in the selection queries there would be no lineage shown since the query could not be parsed properly. This could lead to a loss of important lineage information on customers' side. The following example shows a simple JSON path expression used in the `SELECT` statement.

```
SELECT raw:owner, raw:OWNER, raw:['owner'], raw:['OWNER'] FROM
    store_data;
```

Join

The last feature we will focus on is the table join. In Databricks SQL there is an option to use so-called `JOIN LATERAL` [32] for joining two tables. In this kind of join the right-hand table is specified as a sub-query and the join condition is specified in the `WHERE` clause of the sub-query.

The following example shows how lateral joins can be used in Databricks SQL.

```
SELECT id, name, deptno, deptname
    FROM employee
    JOIN LATERAL (SELECT deptname
                    FROM department
                    WHERE employee.deptno = department.deptno);
```

Showing which tables were combined is a really useful feature and we would like to be able to provide this in the result graph.

4.9.2 SQL scanner

In the previous section, we discussed the differences between the Hive SQL and Databricks SQL dialects. In this section, we will discuss what options are there for the notebook SQL analysis and which is the best one to be used.

The first option is to use the already existing Hive scanner available in Manta as was planned from the beginning. The advantages are that the scanner is already developed and available in Manta hence no additional work needs to be done. On the other hand, there are many differences between the two dialects so we would possibly not be able to analyze all of the customer's codes.

The second option is to take the current Hive scanner and add support for the Databricks SQL-specific constructs. This solution has the advantage that it would be only an extension and there would be no need to spend a lot of time on the configuration of the scanner. The huge disadvantage of this approach is that by mixing the two dialects together we could potentially create a mess in customers' lineage since the scanner would try to produce the lineage for concepts that are illegal in Hive. So instead of an error message that the user has incorrect input, they could get lineage that would not correspond with Hive standards.

The third and final option is to create a separate scanner for Databricks SQL and integrate it into the Manta platform. The advantage of this solution is that the scanner would be specialized in one technology only. Another advantage is that the scanner would have to be designed from scratch so it could be adapted for some Databricks-specific constructs from the beginning and would be easier to extend in the future. A disadvantage of this solution is that it takes a non-trivial time to develop a new scanner and properly integrate it into the Manta platform. Hence there would be higher time requirements for the project.

After discussions with the consultants from Manta, we have decided that the third option is the most reasonable one. Merging Hive and Databricks SQL together would cause confusion and would need several workarounds in order to adapt the scanner properly. Due to this, a decision that a new scanner needs to be developed has been made. However, since it is a huge task, this new Databricks SQL scanner will not be the subject of this thesis. In our solution, we will use the developed Databricks SQL scanner as a black box that has been delivered to the Manta platform by Manta employees.

4.10 Summary

Now that we took a deeper look at all the challenges, concepts and issues let us briefly summarize the goals for the design of the scanner.

In sections 4.1 we concluded that there are two major groups of data that need to be extracted. The data entities and source codes. Section 4.2 showed possible ways for extracting the source codes and related entities like files stored in DBFS, jobs used for scheduling notebooks, and queries. Then in Section 4.3 we described possible ways for extracting information about metastores, catalogs, schemas, and tables. We showed, that the concept can be used by writing a small prototype described in Section 4.4.

As we found out, the Unity Catalog API cannot be used for instances that do not have Unity Catalog enabled. Those instances can use only Hive metastore for storing the metadata. In Section 4.5 we described how we can access the Hive metastore using the JDBC driver provided by Databricks.

Then we moved to the languages used in scripts and how to analyze them. In Section 4.6 we picked the best source format for exporting the notebooks using the REST API. Related to the notebooks themselves, we discussed a very important feature of Databricks - the spark context for notebooks. In Section 4.7 we described how it can be used, and why we need to take it in mind in our scanner as well.

Since Manta supports multiple technologies, databases, and BI tools among them, we decided to take a closer look at the option of connecting Databricks

to external sources. In Section 4.8 we described how Databricks connects to the databases such as Oracle and PostgreSQL.

The last thing that needed to be analyzed more deeply was the option of using an existing Hive scanner to analyze the SQL cells. However, how we found out in Section 4.9 the differences between Hive SQL and Databricks SQL are too extensive, and hence a new SQL scanner had to be developed by Manta employees.

Hence the main goals for the design are:

1. Design the extractor part of the scanner to use the API methods from the REST API analysis.
2. Design a way to support exchanging information between different cell languages.
3. Design the algorithm for the notebook analysis that uses external scanners (Python and Databricks SQL scanners) to analyze the notebook body.
4. Merge the results from the external scanners and Unity Catalog lineage information into one result graph that can be displayed to users.

5. Design

In this chapter, we focus on describing our design of solutions for problems and goals we uncovered during the analysis.

5.1 Scanner design

The first task that needed to be solved was the overall design of the scanner, the components it should have, and how they should interact with each other. Figure 5.1 shows the standard Manta scanner architecture applied on the Databricks scanner use-case.

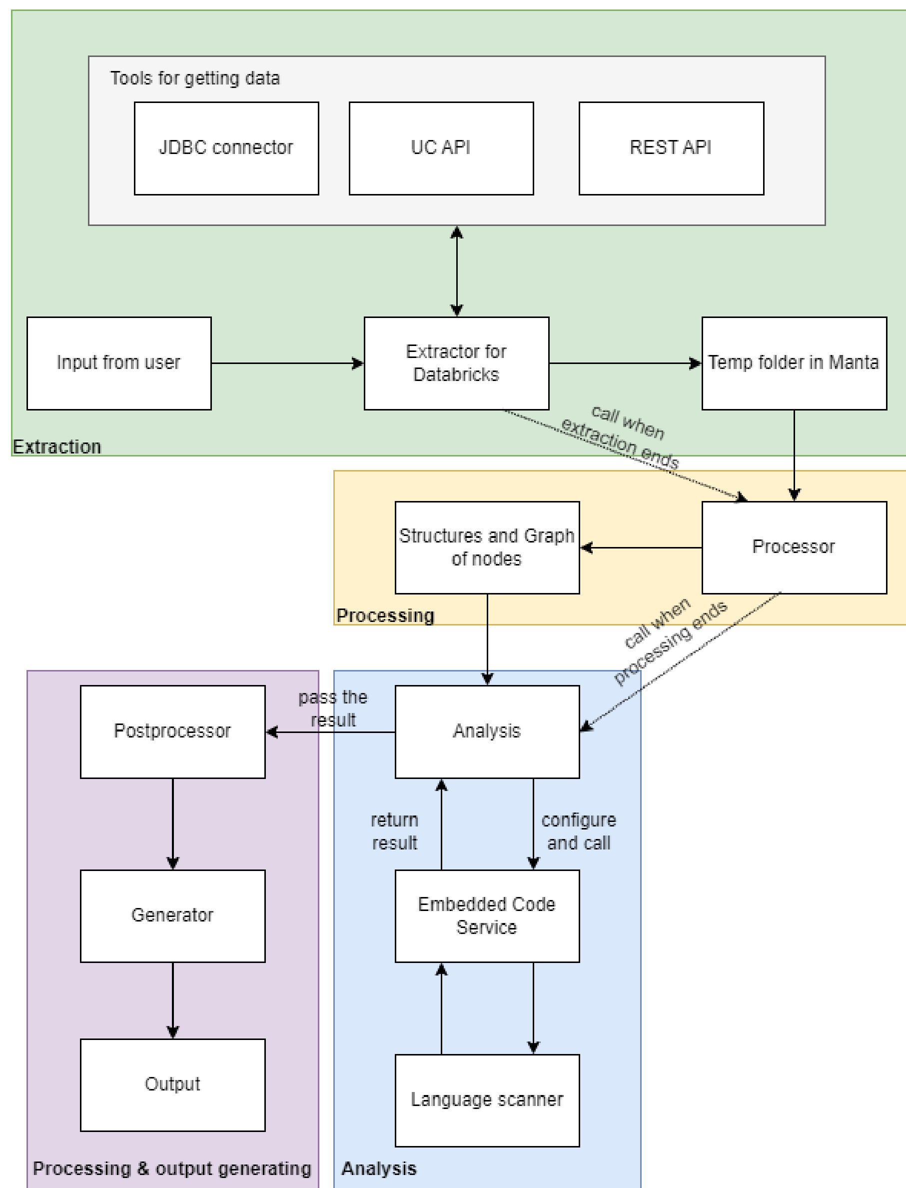


Figure 5.1: Standard Manta scanner architecture applied on Databricks use-case

Let us now describe each component in more detail. The first component is the *Extractor* shown in the green rectangle in the Figure. The extractor is

responsible for downloading all data necessary for further analysis. In the case of Databricks, the Extractor is responsible for getting information about tables, schemas, and catalogs from all available metastores and downloading the source codes of notebooks, jobs, and queries present in the Databricks user workspace. To do so, the extractor uses tools that Databricks provide such as the REST API, Unity Catalog API, and the JDBC connector to Hive metastore.

Once all data are extracted, they are stored in some temporary folder in Manta and can be further processed. That is the job of a so-called *Resolver*. The Resolver takes the raw extracted data and transforms them into a unified model that is further used in the analysis. Once all the data are transformed to a suitable form, they are again stored in a folder and prepared for analysis.

The *Analysis* module takes the transformed data and performs a static analysis of the source codes. In our case, the analysis consists of using the ECS for Python and Databricks SQL to obtain lineage information about cells and eventually for the whole notebook. Once the analysis is finished, the results are sent to *Post-processor*. Post-processor transforms the result to a suitable form that can be passed to the common *Generator* which is responsible for producing the output graph. In our case, the post-processing consists of properly merging the results of different scanners together into one graph object. This graph is then processed by Generator and transformed into a Manta graph that can be viewed in the application.

5.2 Information extraction

Now that we know the overall layout of the scanner let us focus on concrete steps in more detail. The first step we want to describe is the extraction of data. One of the goals of this thesis is to implement the Hive metastore information extraction. However, since the whole Databricks scanner has to combine both Unity Catalog and Hive metastore extraction a unified model has to be used.

The first step is to extract the raw data. In the case of the Unity Catalog extraction, the high-level workflow could work like this:

1. Extractor uses the Unity Catalog API to extract all information about available tables, schemas, and catalogs. Also, the lineage information for tables and their columns can be extracted.

2. Extractor uses the REST API to download notebooks, jobs, and queries.

3. All extracted data are stored in a temporary folder.

For easier imagination of the workflow please see Figure 5.2.

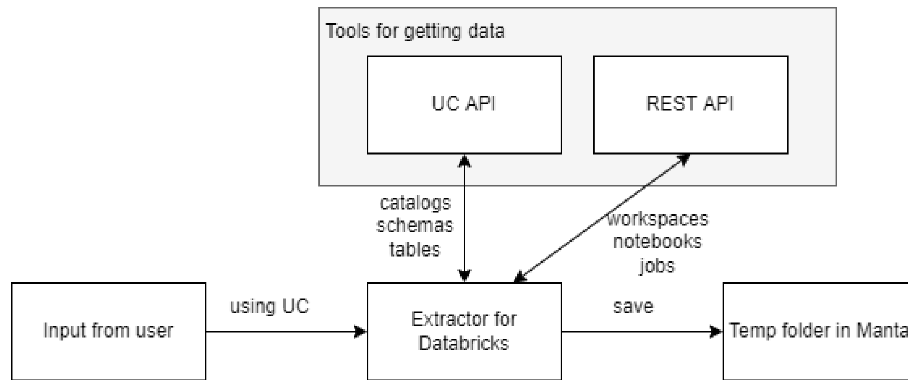


Figure 5.2: Unity Catalog extraction workflow

When it comes to the Hive metastore extraction, the workflow goes like this:

1. Extractor uses the JDBC connector from Databricks to get information on tables and schemas. Since Hive metastore has no catalogs, no information can be extracted about them. Also, Hive metastore does not provide lineage information of any kind, so they cannot be extracted either.
2. Extractor uses the REST API to download notebooks, jobs, and queries.
3. All extracted data are stored in a temporary folder.

For a visual representation of the workflow please see Figure 5.3.

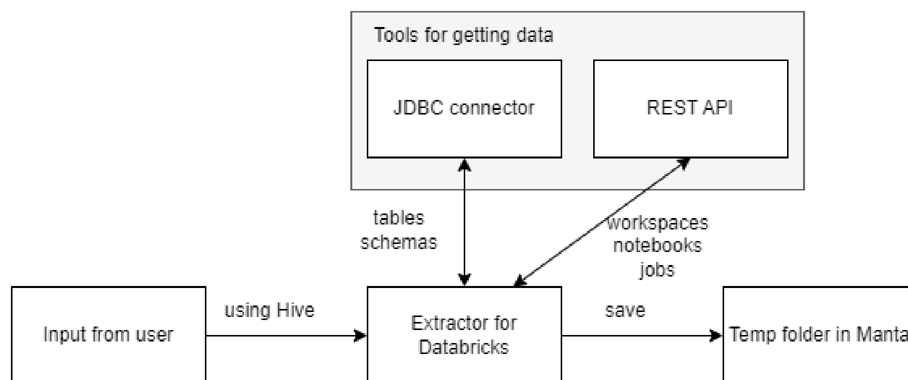


Figure 5.3: Hive metastore extraction workflow

Using these workflows, all of the necessary data can be obtained from the Databricks instance. However, there are some differences between the data that we can get from the Unity Catalog and Hive metastore. Due to this, we need to transform the data into a unified model. For example, since Hive metastore does not have any catalog layer, we need to create one artificially. We know, that Databricks did that as well when they introduced Unity Catalog. The way they did it was by adding the `hive_metastore` catalog and putting all Hive metastore data there. This is exactly what we need to do as well. For each table obtained from Hive metastore a `hive_metastore` catalog name will be added.

To have a unified way of working with data, there is an interface for each data entity that is extracted. It does not matter from which metastore the data are,

all of them are saved in a way that is compliant with the model interfaces. The entities with their own interfaces are:

- catalog
- schema
- table
- table column
- table-level lineage information
- column-level lineage information
- notebook
- notebook command (cell)
- workspace
- query

All of these transformed data are then stored in a so-called repository folder. The structure of the repository folder is shown in Figure 5.4. To have the hierarchy represented in code there is a Repository class that stores information about the transformed extracted data. All of the data stored in the repository folder are prepared to be used by the analysis.

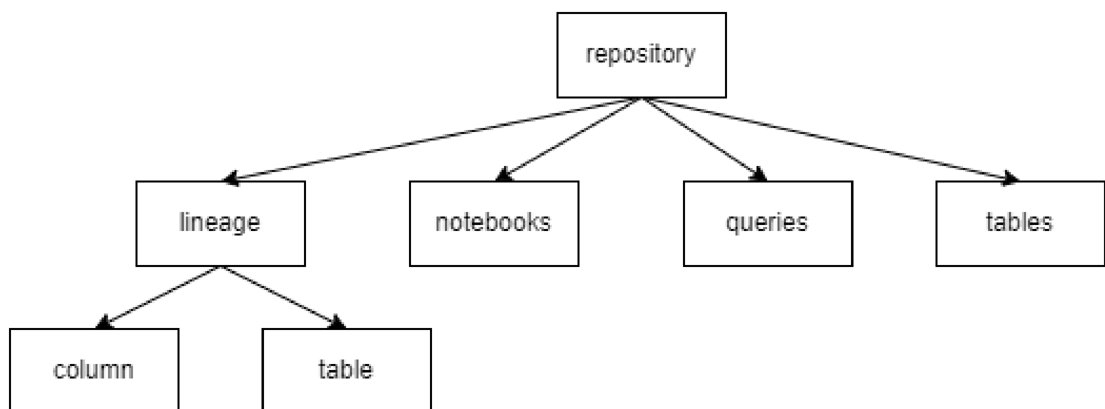


Figure 5.4: Repository folder structure

5.3 Context information in notebooks

There are technologies that support writing scripts, notebooks, or other kinds of source codes in multiple languages. With this being possible, users can use the language that fits the use case the best. As a result, we can get a source code that combines for example Python, SQL, and Scala. Technologies that support this are for example Databricks, Jupyter Notebooks, and to some extent Talend. Since we would like to be able to analyze these kinds of source codes, we had to come up

with a solution for how to pass information between different languages that need to be analyzed. The solution for this problem is to create a shared context that would be passed to scanners and would contain all information necessary for other scanners. In addition to that, each language will need to have its own context so that the information about already analyzed parts of the code (like variables and their values, imported libraries, etc.) would be available when analyzing another part of the source written in the same language.

To sum up, we need to have two kinds of context - shared and language-specific. The shared context has to be more general since it has to be a sort of a bridge between different technologies used by Databricks (or other notebook technology such as Jupyter) whereas the language context is designed specifically for a given language in order to store as much information from analysis as possible so that we would get better results from future analysis of cells.

The following sections provide detailed insight into both kinds of contexts and their usage workflows.

5.3.1 Shared Context - Sharing data between scripts in different languages

This section describes the design for a context used for scanners to be able to share values and tables with each other just as they can in Databricks (and other scanners). Based on the fact, that cells can be written in different languages and can share information between them, we need to have a so-called *shared context* which is responsible for holding information about the values that can be passed from one language to another. An example of a situation where this context is needed is shown in Figure 5.5, where the first cell is written in Python language and stores values into the spark context. The second cell is written in Databricks SQL and uses the values in the query.

Cmd 2

```
1 %python
2 age = 31
3 name = "Ross Geller"
4 spark.conf.set("myapplication.name", name)
5 spark.conf.set("myapplication.age", age)
6
```

Command took 0.04 seconds -- by natalia.potocekova@getmanta.com at 11/8/2022, 6:24:45 PM on Natalia Potocekova's Cluster

Cmd 3

```
1 INSERT INTO friends values ('${myapplication.name}','${myapplication.age}')
```

(4) Spark Jobs

Table

	num_affected_rows	num_inserted_rows
1	1	1

Showing 1 row. | 13.19 seconds runtime

Command took 13.19 seconds -- by natalia.potocekova@getmanta.com at 10/24/2022, 2:29:44 PM on Natalia Potocekova's Cluster

Figure 5.5: Shared context properties example in Databricks

5.3.2 Language Context - Sharing data between scripts in specific language

However, there is a need for one more type of context. Imagine a situation shown in Figure 5.6. There are two cells written in the Python language. The first cell contains all of the imports necessary for the work, the second cell does some part of the work using the imports from the previous cell.

Figure 5.6: Language context example

If we only analyzed the notebook cell by cell, the first cell would not produce any lineage since it's only imports. The second cell could generate some lineage, however, we wouldn't have information about what libraries were imported and the analysis would end up in error. This is exactly the reason why we need to have a so-called *language context*. The language context holds information necessary only for one particular scanner in order to have a complete set of information for analysis of the following cells (such as imports, variables, etc.). This context is specific for each scanner since each technology needs different things for its analysis. This is the biggest difference between the two kinds of context we need to use. The shared context has to be more general since it has to be a sort of a bridge between different technologies used by Databricks (or other notebook technology such as Jupyter) whereas the language context is designed specifically for a given language in order to store as much information from analysis as possible so that we would get better results from future analysis of cells.

Now that we understand the concept of the language context and what it is used for let us design the language contexts for Python and Databricks SQL languages. In the following sections, we introduce a solution for each language separately.

Python local context

The initial idea was to use the Python scanner's internal structures to represent the current state of the analysis. However, that would mean that we would have to put large structures/objects from the Python scanner to the context which did not seem like a good idea. That's why another proposed solution was to have a context that would remember all previously analyzed cells (source codes) and

the latest result graph produced by the analysis. Then when a new cell should be analyzed, the previous cells would be stocked before the cell and the analysis would run on all of them together. The previous cells would be analyzed again, however, no context information would need to be inserted in the scanner since all of the contexts would be analyzed again. This solution is slower but does not require any changes in the Python scanner and hence the scanner can be used as a black box. One thing that would be problematic is that the values in the global context cannot be stored as single values. To demonstrate the problem let us have the following situation:

```
%python
# some other code in the cell
spark.conf.set("var1", "foo")
# some other code in the cell
-------------------------------------------------------
%scala
# some other code in the cell
spark.conf.set("var1", "bar")
# some other code in the cell
-------------------------------------------------------
%python
# another python cell that needs to be analyzed
```

We have 2 cells. The first one in Python the second one in Scala. Both of the cells write the value into the `var1`. Now when the third cell also written in the Python language would be analyzed, the value of `var1` would change from `bar` to `foo` again which is not correct. That's why we would have to store a set of all possible values for each variable. This again leads to imprecise information in the result graph. On the other hand, no extensive changes are required for the Python language local context (or any other language scanner for this matter).

Other resources that could be in the local context are the Databricks libraries. These would be added to the code that should be analyzed.

Databricks SQL language context

When it comes to Databricks SQL there are not many language-specific features that need to be remembered in the language context. Only the value of the current schema and catalog need to be stored so that other languages would be able to work with proper values. Hence all we need to do for this context is to store a set of possible values for both the default schema name and default catalog name.

5.3.3 Context Usage Example

Now that we defined what kinds of contexts are needed let us demonstrate how the communication between the Databricks scanner and other scanners would work using the contexts. Figure 5.7 shows the passing of the context information for Python and Databricks SQL languages.

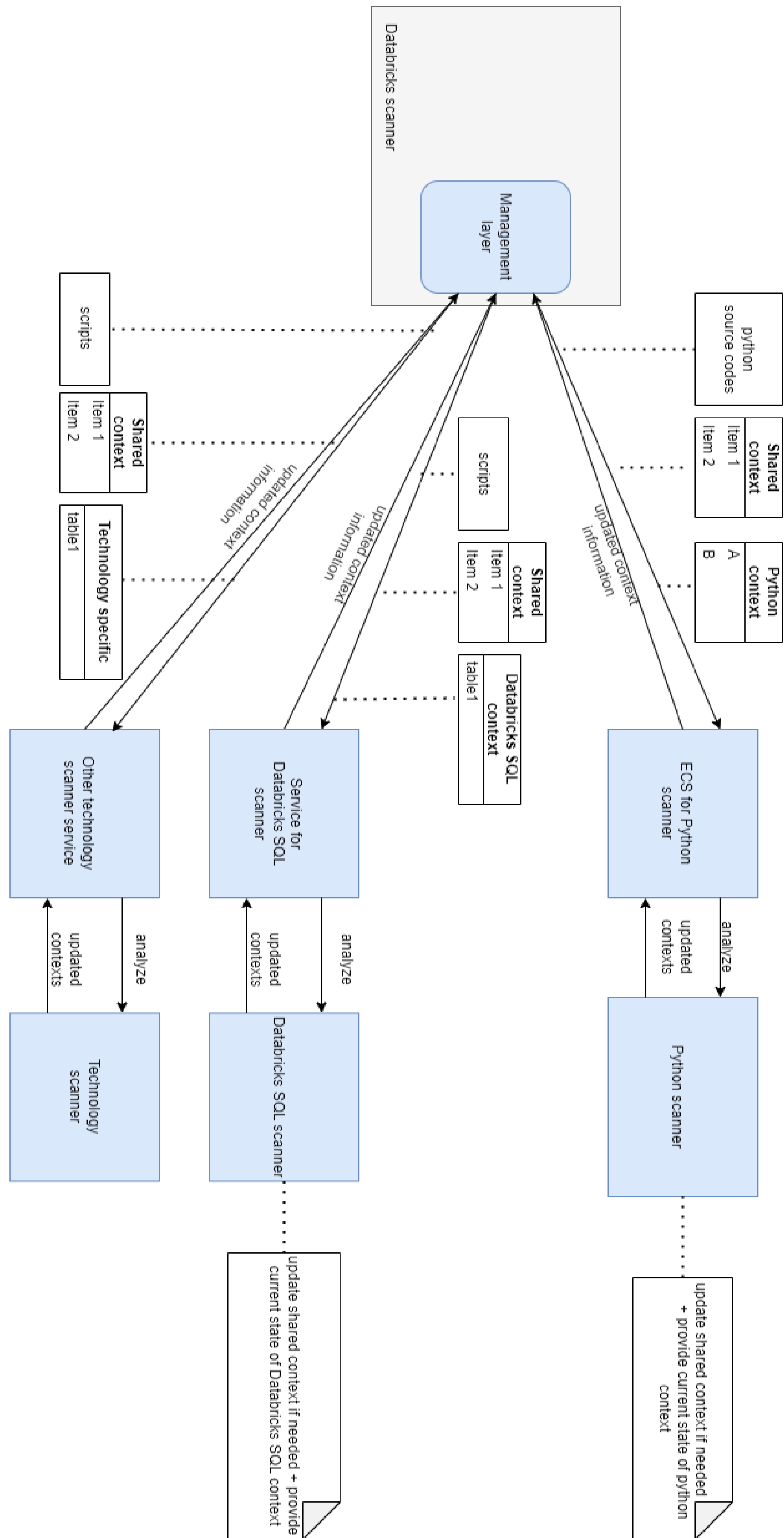


Figure 5.7: High-level communication diagram

Let us now describe the components in the diagram in more detail.

The management layer is responsible for managing the contexts during the analysis. Hence it needs to know what languages are supported for analysis, what source codes are available for analysis, needs to remember the already analyzed results, and should anything be missing in analysis (such as some library or another source code referenced in currently analyzed source code) the management layer should try to find it and provide it for analysis. Should the management layer run out of resources and there is still something missing the missing file name should be stored in some sort of error log so that the user would be informed about it. When the analysis of all source codes and cells is finished, the results will be passed for processing to another component in the scanner.

The Embedded code service (ECS) for Python scanner is responsible for setting up the Python scanner and running the analysis on the inputs it got. In our case the ECS will get context information and cell for analysis, then runs the Python scanner. When the Python scanner finishes execution, results are returned to the management layer alongside updated context information. Then the management layer updates contexts according to the results of the cell analysis and obtained context information.

The service for Databricks SQL will be really similar to ECS but not as general since it will be done for only one language. We have decided to include this service so that the work with embedded code scanners would be the same for intermediate languages and all other languages. Hence this service will be responsible for taking the context, setting up the Databricks SQL scanner, running the analysis, and then returning the results of the analysis back to the Management layer.

To preserve a united way of working with embedded code scanners, each language needs to have its own service that will be responsible for setting up the scanner based on the context. For intermediate languages it will be an ECS implementation, for other languages it will be similar to Databricks SQL service. In the diagram, this is represented by the *Other technology scanner service* component.

5.3.4 Shared context design discussion

In this section, we focus on the design for the shared context part of the assignment. Before the design can be proposed, we need to know what exactly can be shared between languages. Based on the analysis of the spark context (see Section 4.7) variable values and dataframes representing tables can be passed using the spark context. Since we needed to design a solution that could be used in a more general manner by any scanner that would need it, we gathered information from the Manta employees which analyzed Jupyter notebooks, and Talend. In the case of Jupyter notebooks, the situation has been really similar to Databricks since both variables and tables could be passed using the context. Additionally, some Jupyter kernel information could be passed as well. In the case of Talend, there is a context present that can be filled with variables and their values.

In our thesis, we focus only on the variables and how to share their values across languages. In the following sections, we describe possible approaches.

Variable in Shared Context

For variables, we need to remember the **name** of the variable and the **value** of the variable. For example, if we have in Python the following string variable `bar = "bar"` we would need to have a record that a variable named `bar` has a value `"bar"`. However, in order to do so, scanners need to be able to work with constants and their values. There are scanners that are capable of this to some extent (for example the Python scanner that has `ConstantFlow` for representing the constants) but there are still limitations for example if the value of some variable is read from an input, we have no possible way to determine the exact value. Due to this, there should be some default `UNKNOWN` value used when we know that some shared variable was used but we don't know the exact value of it.

Here can be a slight problem with multiple possible values. For example, if we had one cell in Python language that sets the variables based on some condition, there would be multiple possible values for the shared variables. Then if we had another cell in SQL that inserts these values into the table, the Databricks SQL scanner needs to consider all possible options. For an illustration see Figure 5.8.

Cmd 2

```
1   %python
2   import random
3
4   num = random.randint(0, 1)
5
6   if num >0:
7       spark.conf.set("myapplication.name", "Bob")
8       spark.conf.set("myapplication.age", 30)
9   else:
10      spark.conf.set("myapplication.name", "Alice")
11      spark.conf.set("myapplication.age", 25)
12
```

Command took 0.07 seconds -- by natalia.potocekova@getmanta.com at 11/29/2022, 6:38:36 PM on Natalia Potocekova's Cluster

Cmd 3

```
1   INSERT INTO friends values ('${myapplication.name}','${myapplication.age}')
```

Figure 5.8: Multiple possible values example

Due to this, we need to find a solution that would be able to handle multiple possible values. The following sections discuss different options we have for this case.

Storing a list of all possible values

The first option we have when it comes to multiple values is that the values would be stored in a list. Hence if for example variable `foo` could have values `"bar"` or `"barbar"` then the record for the variable would be something like this: `foo = ["bar", "barbar"]` (we used a sort of Python notation to denote that the values would be stored together in one list). In this case, we would need to have only one context that would remember all of the possible values in the list. However, there would be a problem that we would not know which values belong together. Let us take a look at an example situation in Figure 5.9.

Cmd 4

```
1   %python
2   import random
3
4   num = random.randint(0, 1)
5
6   if num >0:
7       spark.conf.set("myapplication.schema", "s1")
8       spark.conf.set("myapplication.table", "t1")
9   else:
10      spark.conf.set("myapplication.schema", "s2")
11      spark.conf.set("myapplication.table", "t2")
```

Command took 0.03 seconds -- by natalia.potocekova@getmanta.com at 12/1/2022, 3:53:07 PM on Natalia Potocekova's Cluster

Cmd 5

```
1   %sql
2   SELECT * FROM ${myapplication.schema}.${myapplication.table}
```

Figure 5.9: Example of variable values context

In this situation, there are two possible outcomes: either the `s1.t1` table should be used or the `s2.t2` table should be used. However, if we only kept the lists of values, we would also have pairs `s1.t2` and `s2.t1` which are not valid. This would lead to over-approximation of analysis results.

To summarize this option, an advantage would be that only one context would be needed. The disadvantage would be that it is an over-approximation hence not valid options are analyzed as well. Also, a slight limitation is that SQL scanners cannot work with sets, so we would have to have some sort of ”manager” that would be responsible for passing the values one by one.

Prototype-like context idea

Another idea we could use for the shared context that considers multiple values would be a prototype-like context design. In the beginning, there would be one empty parent context created. This context would be sent to the scanner that needs to analyze the first cell of the notebook. In case some branching occurs (if-else, loop, etc.) a child context would be created from the parent context (cloning) and the possible shared context values would be written there. Once the branch would end, the current context would be set back to the child's parent. In case some change occurs in the parent context after some child has already been created, the change needs to be propagated to the child as well. Using this pattern we would get a list of contexts with only possible valid options (we would avoid the Cartesian product in the first mentioned solution). On the other hand, we would need to work with multiple possible contexts and call other scanners on all of them. However, since it is not usual to have tons of shared variables in scripts (since scripts in Databricks are usually single-purposed) we do not mind taking care of more contexts if we could in the end reduce the number of callings of other scanners and improve the accuracy of the analysis.

Since this concept is not as easy to imagine as the previous one, let us demonstrate it in an example shown in Figure 5.10.

Cmd 4

```
%python
import random

num = random.randint(0, 1)

if num >0:
    spark.conf.set("myapplication.schema", "s1")
    spark.conf.set("myapplication.table", "t1")
else:
    spark.conf.set("myapplication.schema", "s2")
    spark.conf.set("myapplication.table", "t2")

spark.conf.set("myapplication.catalog", "c")
```

Command took 0.03 seconds -- by natalia.potocekova@getmanta.com at 12/1/2022, 3:53:07 PM on Natalia Potocekova's Cluster

Cmd 5

```
%sql
SELECT * FROM ${myapplication.catalog}.${myapplication.schema}.${myapplication.table}
```

Figure 5.10: example of multiple contexts due to branching

In the beginning, an empty first context would be created. This context would be sent to the analysis. Let us now have the following situation. In this case, the workflow with context could look like this.

1. On lines 2-5 the original context is as it was

2. On line 6 there is an if statement. This would lead to the creation of a child context. The workflow should be something like this (the # comments will symbolize what would need to happen):

```
%python
if num > 0:
  # here call the context manager to create a child context
  spark.conf.set("myapplication.schema", "s1") # write to the context value "s1" for variable myapplication.schema
  spark.conf.set("myapplication.table", "t1") # write to the context value "t1" for variable myapplication.table
  # here call context manager to switch back to the parent (end of branch section)
else:
  # here call the context manager to create a child context
  spark.conf.set("myapplication.schema", "s2") # write to the context value "s2" for variable myapplication.schema
  spark.conf.set("myapplication.table", "t2") # write to the context value "t2" for variable myapplication.table
  # here call context manager to switch back to the parent (end of branch section)

spark.conf.set("myapplication.catalog", "c") # write to the context value "c" for variable myapplication.catalog

------------------
%sql
# Input for the SQL scanner: global Databricks context, local SQL context, text of SQL command with unresolved references to variables ${...}
```

```
# Contexts include all (valid) combinations of values for
    the referenced expressions (catalog, schema, table)
# SQL scanner need precise inputs (not just one big set as
    Python/Java scanners)
SELECT * FROM ${myapplication.catalog}.${myapplication.
    schema}.${myapplication.table}

```

For this situation the contexts would look like this:

1. Original first context

```
myapplication.catalog = "c"

```

2. Context for the `if` block

```
myapplication.schema = "s1"
myapplication.table = "t1"
myapplication.catalog = "c"

```

3. Context for the `else` block

```
myapplication.schema = "s2"
myapplication.table = "t2"
myapplication.catalog = "c"

```

All of these contexts would be returned after the end of the analysis. As can be seen in this example, the values that were written to a parent context are also propagated to the child contexts.

As mentioned before, this could lead to a potentially large number of contexts. For example, let us take a look on the following situation:

```
%python
if condition1:
    spark.conf.set("myapplication.schema", "s1")
    spark.conf.set("myapplication.table", "t1")
else:
    spark.conf.set("myapplication.schema", "s2")
    spark.conf.set("myapplication.table", "t2")

if condition2:
  spark.conf.set("myapplication.catalog", "c1")
else:
  spark.conf.set("myapplication.catalog", "c2")
```

In this case, we would need to have five contexts since we have one parent context and then 4 branch possibilities. Figure 5.11 shows the division of contexts.

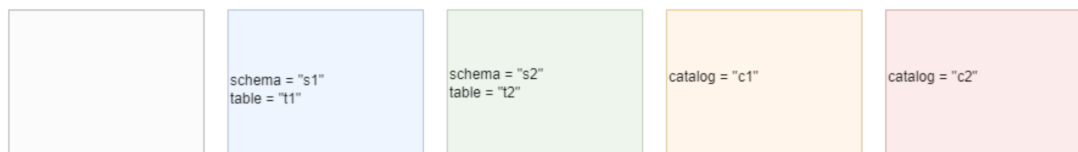
```
if condition1:
    spark.conf.set("myapplication.schema", "s1")
    spark.conf.set("myapplication.table", "t1")
else:
    spark.conf.set("myapplication.schema", "s2")
    spark.conf.set("myapplication.table", "t2")

if condition2:
  spark.conf.set("myapplication.catalog", "c1")
else:
  spark.conf.set("myapplication.catalog", "c2")
```

Figure 5.11: Larger number of contexts example

Here is one problem. Not only there is a large number of contexts but the changes that happened in the second if-else statement need to be somehow propagated to the first two contexts of the first if-else statement. if they did not, we would have the following contexts shown in Figure 5.12.

Context values:



Context hierarchy:

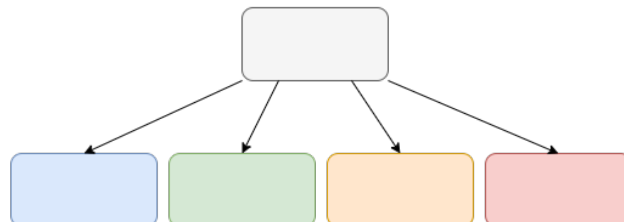


Figure 5.12: An example of possible contexts

However, this situation does not contain context with the correct possible options since we have four of them right now:

- `c1.s1.t1`
- `c1.s2.t2`
- `c2.s1.t1`
- `c2.s2.t2`

None of these options is present in any of the created contexts. So what would have to happen is that four additional contexts would have to be added that would be cloned from the already existing children. The situation is shown in Figure 5.13.

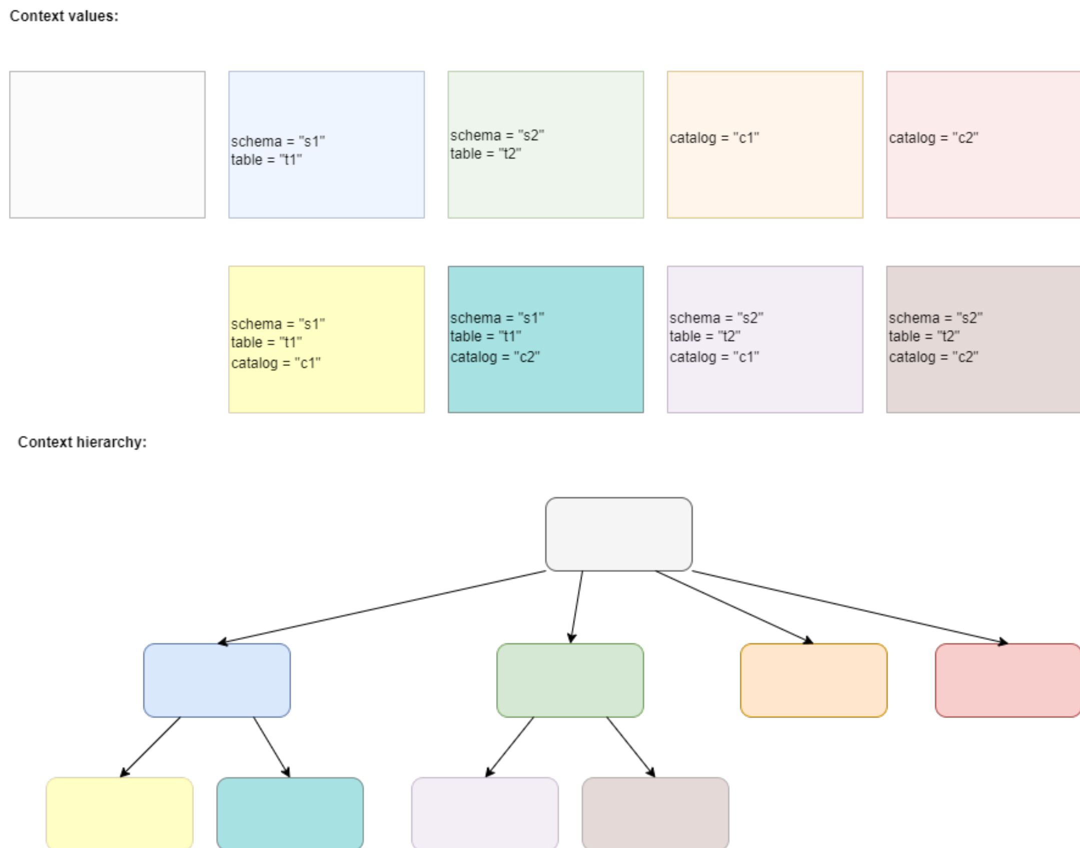
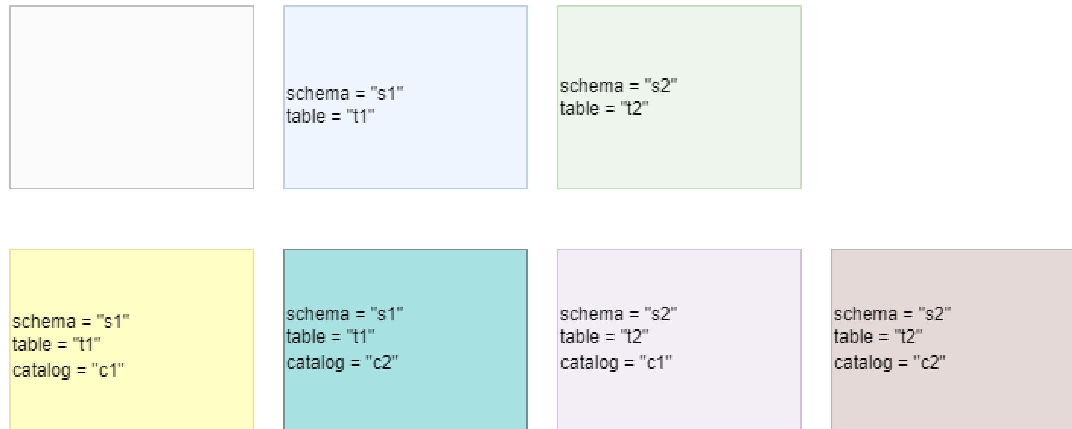


Figure 5.13: An example of all needed contexts

However, what we can notice in this situation is, that the first if-else statement fully covers all of the options (in other words we know that one of the branches needs to be executed). Hence we can do a little optimization here and create copies of context only from the blue and green ones. Then the situation would look like shown in Figure 5.14.



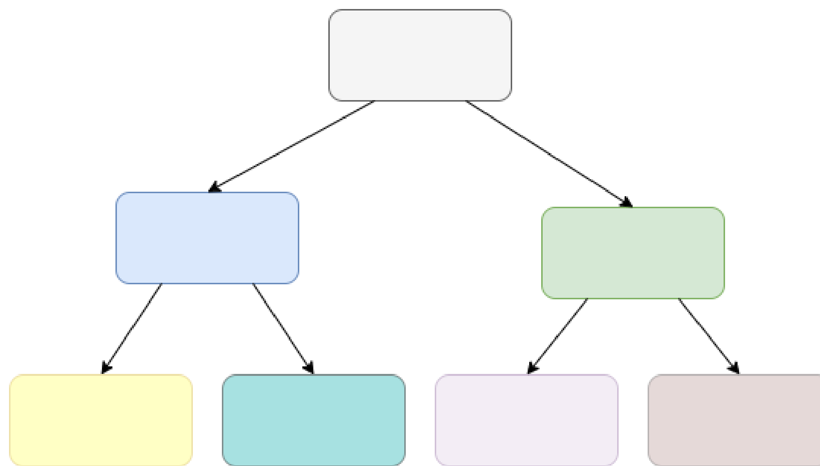


Figure 5.14: An example of reduced contexts

Thanks to this, we would have only seven options for context, where one is empty (the parent context) so it could be reduced, and then six possibilities remain. Since the last four contexts contain everything that their parents contain and more, we would take only these contexts for further analysis in other scanners. So that means, that from the Cartesian product we reduced to four callings of the SQL scanner in this situation. Which is way better.

An important note here is, that this approach would be used only for the variables in a shared context. Usually, scripts do not contain many of those, that's why we don't mind taking care of multiple contexts because, in the end, it would lead to fewer scanner callings.

To summarize this option, the advantages would be that we get more accurate results and a lower number of options that have to be analyzed. On the other hand, we would have a larger amount of contexts to work with and all scanners that would work with this kind of context would have to be adapted to the concept, which can be really time-consuming.

5.3.5 Shared context - final design solution

In order to have a somewhat functional solution in some reasonable time we cannot afford to implement hard and time-consuming solutions as a part of the thesis. Hence in this design, we will focus on finding some compromise between difficulty, efficiency, and time consumption. The approach we would like to take is to have a working solution even though it would be slower at first and then later to optimize it to be better, more precise, and faster. Now that we have stated the rationale behind the design, let us describe what the context would look like.

The shared context needs to store information about variables - their names and values and tables/temporary views created through a spark from dataframes - their names and a pointer to their node. Should some other scanner later in the future need something else, the context can be easily extended to store information about anything they will need.

The workflow for the shared context would look something like shown in Figure 5.15.

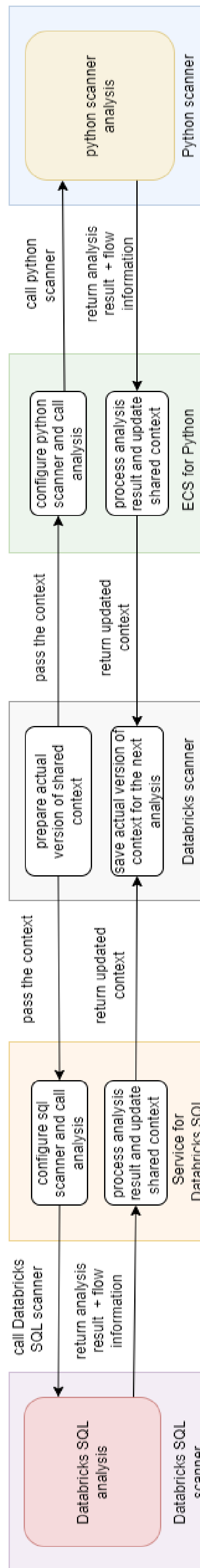


Figure 5.15: Shared context workflow

In order to provide some sort of unified way of working with the context and in order to forbid uncontrollable modifications of the context, the context has its own interface for adding and updating values of variables or tables/views.

As we mentioned earlier, the variables need to have stored their names and values (or the default value `UNKNOWN` if we don't know the value precisely). For the tables, the context stores the table/view name and the node that represents the table so that the table can be matched later in the graph. When it comes to the variable values, we will have to keep all possible values of the variable in the context as a set. This is due to the fact that for now, the proposed solution of analyzing code cells of the same language consists of repeating the analysis since we have to put the previously analyzed cells into the current cell (please see the Section Python local context for more detail).

Another issue we needed to discuss is, how to write the variable values into the context. There are several options for this situation. The first option is to modify the Python scanner extensively and write the values in propagation modes. Another option is to extend the flow summary that the Python scanner produces in their worklist algorithm to contain the variables/tables flow information and the ECS would take the information out of the summary and put it into the context. In order for ECS to know which variables should be written into context there would have to be some sort of flag that would represent that the variable is in the shared context. The last but definitely not least option is a sort of compromise between the two already mentioned approaches. There would be two kinds of objects, a so-called `Insight` and `Outsight`. These two classes would be immutable. The Insight would contain all relevant values that were found during the analysis of the Python cell. The Outsight on the other hand would contain any information that the Databricks scanner needs to provide to the Python scanner before the analysis begins like the already known variable values. The values that should be written into the Insight will be written there in the propagation mode that is responsible for handling the spark methods that set values into the context. The Outsight values would be taken out of the Outsight in propagation modes that handle spark methods that get values from the spark context. Then after the analysis, the Databricks scanner would update the shared context based on the values in the Insight.

The option with the Insight and Outsight objects is the best trade-off between changes in the Python scanner, the generality of the solution, and easiness of use. However, this option would not solve the issue with over-approximation. Since the variable values would be written in propagation modes, we would not have the information on which variables values are valid combinations. However, since Databricks notebooks do not contain that many shared variables, the over-approximation is not that big of an issue here. Hence, we selected the Insight and Outsight approach as the most suitable option.

Notebooks calling each other

In Databricks users can use their custom libraries that can be imported in notebooks and also can call another notebook from the current notebook. This can result in a chain of callings between notebooks. From what we tested, each notebook has its own spark context, so the only way for notebooks to pass values to each other is the input variables and the `returnValue` which stores the output

value of the notebook.

In this section, we discuss how these situations could be handled with respect to the optimized design that was described in the previous section. As of now, the details will focus on the Python scanner mostly, since it is the subject of this thesis, however, these thoughts can be applied to any other scanner that will be used in the future for Databricks or any other notebook technology analysis.

In Python, notebooks can call each other using the `dbutils.notebook.run` method. Let us demonstrate this concept with a simple example. Let us have a simple notebook called `notebookA` which defines values for catalog, schema, and table names. Let us also have another notebook called `notebookB` which takes catalog, schema, and table name as a parameter and creates a new table catalog.schema.table. Then we could call the `notebookB` from `notebookA` in a following way:

```
1 dbutils.notebook.run("notebookB", 60, {"catalog": catalog, "
    schema": schema, "table": table})
```

In Python scanner method callings are usually handled by propagation modes. However, in this case, it is not possible to properly replicate the data flow in the propagation mode, since we need to call the analysis of another notebook first. The solution to this would be to extend the shared context to contain a list of notebooks that need to be analyzed with the arguments they take as input. The propagation mode created for handling this situation would note the notebook name into the Insight object (mentioned in the previous section) and the Databricks scanner would then update the shared context accordingly.

Then when the Databricks scanner would see, that the updated context contains a list of notebooks that need to be analyzed, it would finish the analysis of the current notebook and after that would start the analysis of the following notebook that is required in the current notebook. We need to keep in mind, that every time we detect a notebook that calls another notebook, the parent notebook needs to be analyzed again since notebooks can share values through input arguments and return values. Another thing that needs to be considered is, to store some sort of flag, that a notebook call was already handled, so that the analysis would not be in an endless loop. This leads to implementing a worklist algorithm over nested notebooks. The complete design of the worklist algorithm can be found in Section 5.5.3.

Concrete example

Let us now demonstrate step by step how communication would look like in real-life examples. Let us have the following situation shown in Figure 5.10.

As can be seen in the Figure, one cell is written in Python, then the second cell is written in Databricks SQL and uses values set by the previous cell. The communication would go as follows:

1. Databricks scanner would prepare new context. Then would create an empty Insight and Outsight (based on the empty shared context) and would pass it alongside scripts to the ECS for Python.

2. ECS for Python would take the information and would set up the Python scanner accordingly.

3. ECS would send scripts to the Python scanner for analysis.
4. When the Python scanner finishes analysis, ECS returns the updated Insight to the Databricks scanner.
5. Databricks scanner updates the context information (shared context with values from Insight and language context with the currently analyzed cell) and takes a look if some error occurred. If not the analysis continues with another cell.
6. Databricks scanner sends the current version of shared context, new Databricks SQL context, and scripts to the Databricks SQL service.
7. The Databricks SQL service will prepare the Databricks SQL scanner accordingly.
8. Since we have more than one option of inputs for Databricks SQL scanner, one option at a time the service would pass the values and scripts to Databricks SQL scanner.
9. For given inputs Databricks SQL runs analysis and returns results.
10. All results are collected by the service, then written to the context, in case something changed in Databricks SQL specifics, the language context is updated as well.
11. When all options were analyzed, updated contexts are returned to the Databricks scanner.
12. Databricks scanner looks if there was any error, if not, then it will end analysis.
13. The results will be passed to the component responsible for composing the final graph.

Figure 5.16 contains a visual demonstration of this workflow in time.

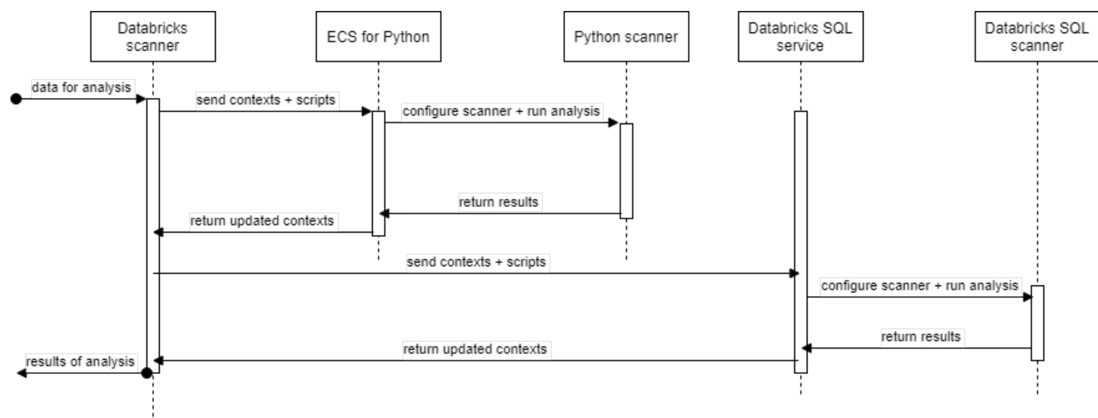


Figure 5.16: Notebook analysis workflow

Another situation that might occur is when one notebook imports other notebooks as can be seen in Figure 5.17.

```
Cmd 1

1   catalog = "c1"
2   schema = "s1"
3   table = "t1"
4
5   spark.conf.set("testVar", "test")

Command took 0.72 seconds -- by natalia.potocekova@getmanta.com at 12/12/2022, 1:41:24 PM on Natalia Potocekova's Cluster
Cmd 2

1   dbutils.notebook.run("notebookB", 60, {"catalog": catalog, "schema": schema, "table": table})
```

Figure 5.17: Notebook calling another notebook example

The situation would be handled in the following fashion:

1. Databricks scanner would send the first cell for analysis to ECS for Python alongside the context information in an Outsight.
2. ECS would set up a Python scanner and send the cell for analysis.
3. Python scanner finishes analysis and returns results.
4. Databricks scanner saves the cell to the local context, and updates shared context accordingly based on the data from Insight.
5. Since there are no issues, Databricks scanner sends second cell to ECS together with current versions of context information.
6. ECS again sets up a Python scanner and runs analysis on all previous cells from context and the current one.
7. When the Python scanner encounters the `dbutils.notebook.run` method it notes into the Insight that the analysis of this notebook is needed.
8. When analysis finishes, the Python scanner returns results.
9. ECS returns the results to Databricks scanner.
10. Databricks scanner updates the shared context based on Insight and sees that notebook needs to be analyzed.
11. Databricks scanner saves the current Python language context and starts the analysis of the second notebook in the same manner as the first one.
12. Once all cells from the second notebook are analyzed (supposed there were no following notebook calls) the analysis of the first notebook takes place again with the updated values from `notebookB`.
13. When the last cell of `notebookA` is finished, the results are sent to the component responsible for combining graphs.

Figure 5.18 shows a UML sequence diagram for this situation.

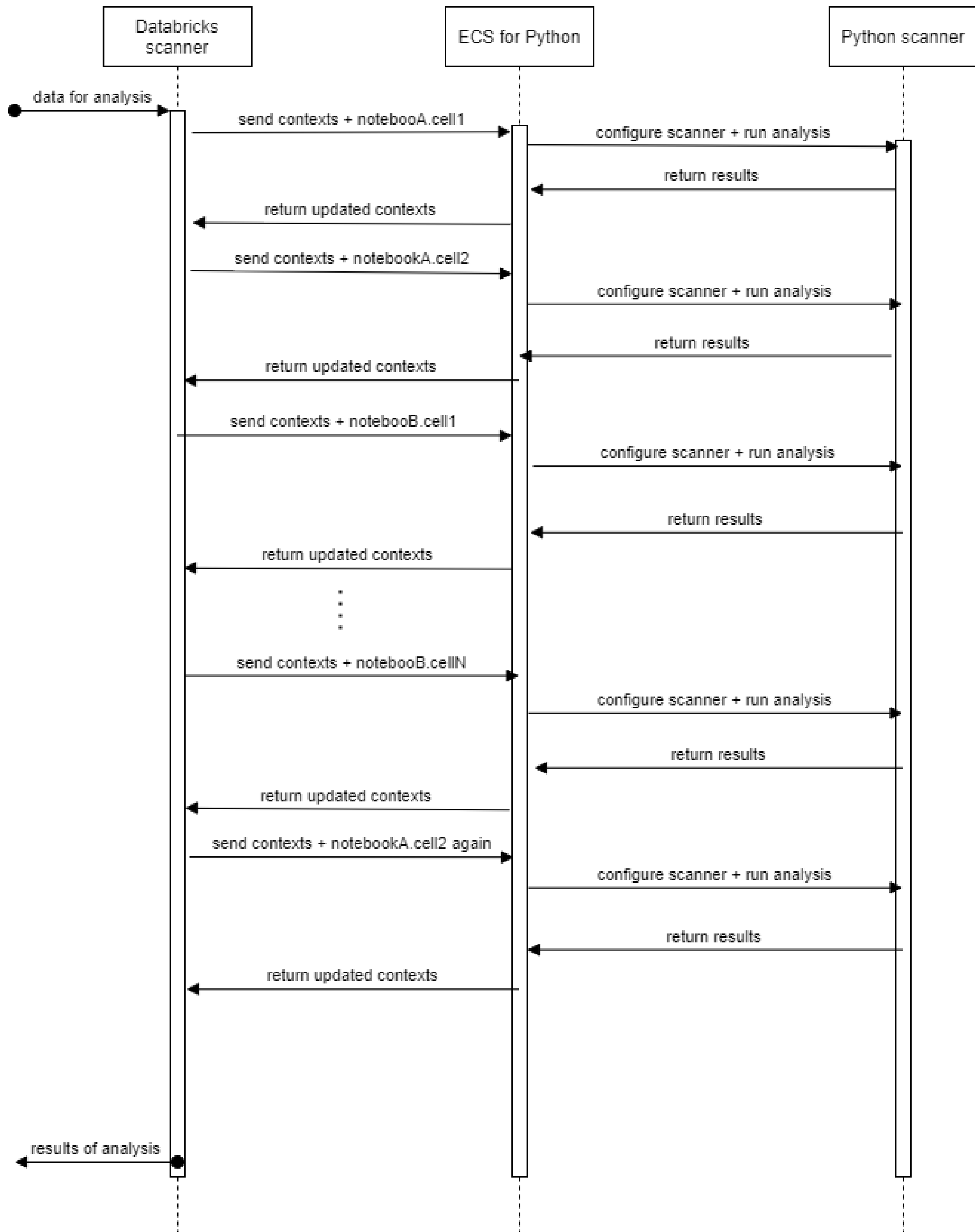


Figure 5.18: Notebook calling another notebook workflow

5.4 Scanners integration

In previous sections, we discussed the challenges related to shared context and how they can be solved. In this section, we describe how the information from the shared context can be passed to Python and Databricks SQL language scanners.

5.4.1 Python

As we briefly hinted in the previous section, in order to pass information to or from the Python scanner we will use the so-called `Insight` and `Outsight` objects implemented by Manta employees. Let us now describe in detail how this idea works.

In situations where a Python scanner is called for analysis of embedded code, there is often a need for passing external information, that is important for the caller technology but does not affect the Python analysis in any other way. To enable this, Manta employees created an interface called `Insighter`. The basic idea is, that the caller technology creates an empty Insighter object and passes it to the Python scanner. During the analysis, the propagation modes that are designed to work with Insigher use the passed object and write information into the Insighter. The important note here is, that the Python scanner does not know about the external technology that is called it. Python scanner only works with the Insighter interface and it is up to the developers of the external technology scanner to implement the Insighter object and related propagation modes correctly. After the analysis ends the Insighter object can be transformed into an immutable object called `Insight` that contains all gathered information from the Python scanner. In our case, this is used for gathering variable values that were written into the spark context via method `spark.conf.set`.

In a similar manner, there is sometimes a need to provide information for the Python scanner (such as variable values) before the analysis begins. To do so an `Outsight` interface can be used. Again, it is up to developers to implement the interface properly and also to implement related propagation modes that use the values provided by external technology and pass them to Python analysis correctly. In our case, this is used for passing all already gathered variable values from shared context to Python scanner, so that when the `spark.conf.get` method is called, the Python scanner would have all available information prepared.

Another important thing related to the Insighter and Outsight are so-called Pin nodes. The Pin nodes are special kinds of nodes used to visualize the flow that would otherwise be skipped or missed. For example, let us have a notebook with two cells as can be seen in the code snippet below:

```
1 # cell 1
2  spark.conf.set("tableName", input())
3  ---------------------------------------
4 # cell 2
5  print(spark.conf.get("tableName"))
```

This is a very simple example where the first cell takes the value that the user wrote on standard input and saves the value into the spark context as variable `tableName`. Then the second cell takes the value of the variable `tableName` from the spark context and writes it on standard output. Without the Pin node support, there would be no flow shown since the Python scanner shows only terminal operations that have some data flow in it. The first cell would be evaluated like this:

1. input() call would be found and an appropriate flow representing it would be created

2. then the `spark.conf.set` method would be found, and it would get the constant `"tableName"` and the flow representing console read as parameters. The propagation mode for this method would note into the Insighter that the value of the variable `"tableName"` is `UNKNOWN` since the input value is not known.

3. There would be no edge present because there has not been any interesting dataflow from the point of the Python scanner.

A similar situation would be for the second cell. However, having the pin node that would represent the write to spark context (and hence to Insighter) and then another pin node that would represent the value present in the spark context (that Python scanner obtained from the Outsight) we would have the situation shown in Figure 5.19.

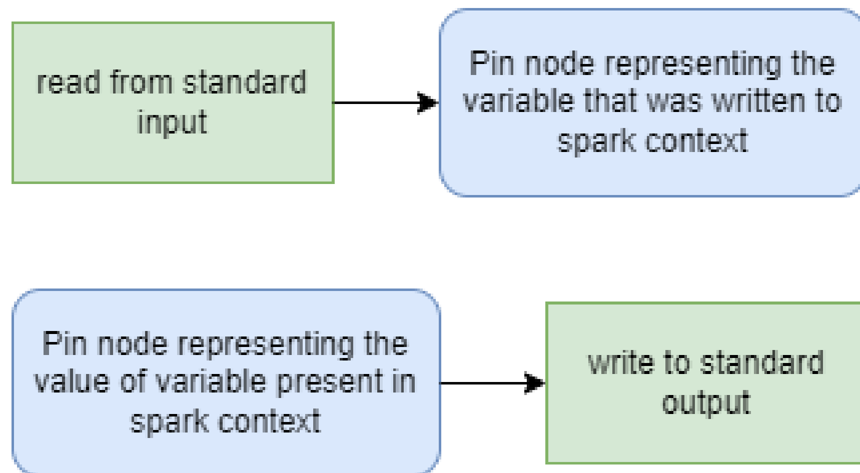


Figure 5.19: Pin nodes placeholders example

Here, we can see at least the information about the fact that some data flow is present in the notebook. However, we don't see that the data read from the input were printed to the output. This can be easily fixed. Each pin node has to have its own name. By naming the input pin node by the name of the variable that was written into spark context and by naming the output pin node by the name of the variable that was obtained from Databricks scanner through Outsight, we would be able to find the input pin and output pin with the same name and connect them with the edge. Then the graph would look like shown in Figure 5.20. The pin nodes are connected by an edge and hence we can see that the data from the input are the ones that were put to the output.

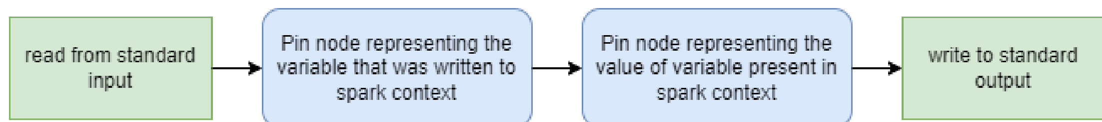


Figure 5.20: Pin nodes connected example

There is one more improvement that could be done in this situation. Combining the two pin nodes into one node that would represent the spark context

itself. Then what we would see is that the information from input was written into spark context and then eventually taken out of it and written to input. Then the graph would look like shown in Figure 5.21.

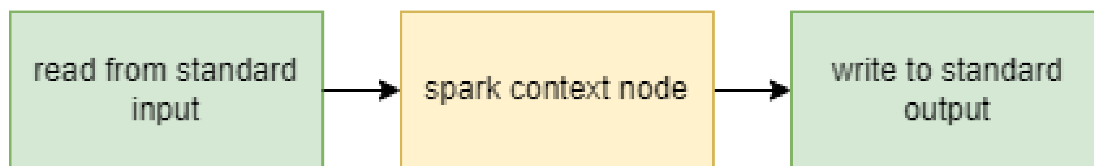


Figure 5.21: Spark context node example

Support for all the actions we described regarding Pin nodes was provided by Manta employees in the Embedded code service implementation. As mentioned in Section 2.3 Embedded code service is a tool that is supposed to be used by external technologies to analyze their embedded Python code using the Python scanner.

To be able to use the ECS for Databricks, the ECS implementation needs to be adjusted. The first step is to define a configuration that contains all information needed to set up the Python scanner properly. In our case, the configuration has to contain the Insighter, Outsight, notebook name, and additional resources such as libraries used in the notebook.

Then we need to define how the source codes and libraries should be stored in the file system so that the Python scanner would be able to work with them properly.

Once these two steps are done, the only thing that needs to be done then is that the Databricks scanner would call the ECS on the Python cells with the configuration that would contain an empty Insighter, and Outsight filled with variable values that are already in the shared context, the external libraries that are available and the notebook name.

In the Python scanner, an Insighter collaborative propagation mode has to be implemented that would handle the `spark.conf.set` method in a way that would write the variable name and value to the Insighter and would create an appropriate input pin flow that would be then transformed into a pin node that represents the variable. Also, an Outsight collaborative propagation mode has to be implemented for the `spark.conf.get` method. This method would take the value from the Outsight for a given variable name, create an appropriate flow that would represent the variable value later in Python analysis, and also create an output pin flow that would be transformed to a pin node representing the variable as input from an external source.

In Databricks scanner, the functionality for connecting the Pin nodes provided by ECS result can be used when merging analysis results into the final graph.

5.4.2 Databricks SQL

Now that we described how to interact with the Python scanner, let us dive into Databricks SQL. As we mentioned in Section 4.9 the original plan was to use the Hive SQL scanner. However, the differences between Hive and Databricks SQL

were too big and a new scanner had to be developed specifically for the Databricks SQL. This scanner had been developed by Manta employees and hence is not part of this thesis. Nevertheless, we use this scanner to analyze the SQL cells from Databricks notebooks.

To do so, we use an Embedded service for the Databricks SQL scanner. This service is really similar to the Python embedded service but it is not as complicated. Embedded code service had to be designed in a way that any new technology can be easily added in the manner described above. Since Databricks SQL is a technology related to Databricks only, the service has been tailored for the Databricks scanner only. This service has also been developed by Manta employees and its code is not a part of this thesis. However, what needs to be implemented is the configuration class (similar to the Python configuration described in the previous section). In our case, this configuration has to contain values of the catalog and schema that are currently considered as default, and all variables with their values we have in shared context since these variables can be referenced in SQL scripts.

Since the possible values of the default catalog name and default schema name can be a set of values (due to branching or any other similar situation) we need to call the analysis on all possible pairs of values. Each result has to be stored since each combination represents a different situation. All of the gathered results will then be merged into the final graph after the analysis ends.

Using the `USE` commands the value of the default catalog or schema can be changed. In order to know the values, we need to modify the existing Embedded code service to store these values in the Embedded code result class. The values are already present in the result from the Databricks SQL scanner, so the only thing that has to be done is to take them and properly store them in the result class of the ECS. Then in the Databricks scanner, the Databricks language context is updated based on these values.

5.5 Notebook analysis

After the successful extraction of information, the analysis takes its place. In our thesis, we focus on the analysis of Databricks notebooks. In the following sections, we describe workflow designs necessary for analyzing notebooks using the external scanners in Manta as well as the algorithms that need to be implemented.

5.5.1 Analyzers

In this section, we describe the design for combining different approaches to notebook analysis. Databricks scanner has to combine the Unity Catalog lineage and the lineage produced by Manta scanners based on the notebook source codes. The approach of using Manta scanners has higher priority than the one that uses Unity Catalog lineage information. The reason behind this is that Unity Catalog provides only information related to tables and columns. Using the Manta scanners we can see also the transformations that were performed with data. However, there are situations when Manta scanners do not produce any lineage (for example when users use constructs that are not yet supported), and due to this, we have the Unity Catalog lineage as a sort of backup plan.

The top-level part of the notebook analysis starts in the Dataflow task class. This class represents an entry point for the analysis. All the extracted data are provided to this class and are then used for the lineage computation. There the analysis using the Manta scanners is performed first and if it fails, the Unity Catalog approach is used instead.

To unify the way of calling different approaches of analysis, easier addition of new approaches, reordering of priorities, or removing some approaches we propose the following design. The Dataflow task has a so-called *dispatcher* that contains a list of *analyzers* ordered by priority. Each analyzer implements a common interface and has method `analyze` that takes analyzer context and notebook information as parameters and returns a Boolean representing the success or failure of the analysis. In the Dataflow task, the only thing that has to be done is calling the `analyze` method on the dispatcher. Inside of the `analyze` method the dispatcher tries to call the analyzers based on the priority. Now, two things can be done - either all available analyzers are called or only the highest priority analyzer is used. This can be set by users depending on what they prefer. The results of each called analyzer are stored separately and are marked by a flag that indicates which analyzer produced the result.

For easier imagination, let us demonstrate the proposed design in our current situation with two kinds of analyzers - one for Unity Catalog lineage, the other for calling external scanners from Manta. Figure 5.22 shows the workflow for this scenario:

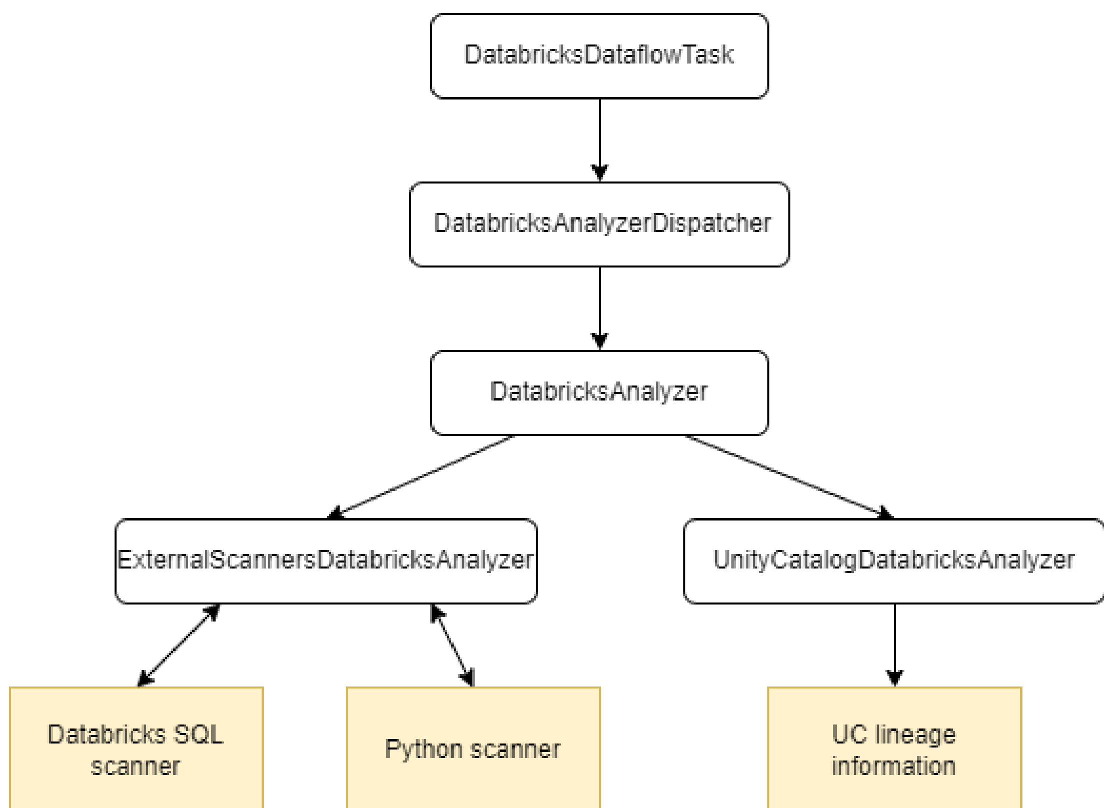


Figure 5.22: Analyzers workflow

Since the subject of this thesis is the analysis of notebooks using external

scanners, let us now describe the analyzer responsible for this approach.

External scanners analyzer

As we mentioned in Section 4.6 Databricks notebooks are basically ordered groups of cells that contain the source codes written in possibly different languages. In this section, we describe how to handle the analysis of a single notebook using the external scanners analyzer. The main idea of the analyzer is as follows:

1. Iterate over notebook cells.
2. For each supported language call the language scanner with proper shared context and language context.
3. Save the results of the external scanners.
4. If the analysis of a cell was successful continue with another one.

Since calling the language scanner service takes a little bit of overhead in terms of data preparation, we decided to create a handler for each language we support. Currently, only handlers for Python and SQL are implemented. However, once Scala and R languages will be supported in Manta, new handlers can be easily added in a similar manner as for Python or SQL. The handlers take the parent node, notebook information, and the cell itself as parameters.

In the case of Python, based on the shared context an Insighter and Outsight are created as we described in Section 5.4. Another thing that has to be done is that cell source code is enriched to contain the definition of a spark context variable. This needs to be done in order for the Python scanner to work properly. Without this, the Python scanner would not know to what object the source code refers when using the `spark` variable. This is caused by the fact, that Databricks initializes this variable internally when running the notebook, hence this step is not present in the notebook source codes.

Then the ECS for Python is called and once results are returned, they are saved to a helper result class. This class contains the notebook name and the result returned from ECS. The Python handler updates the ECS result to contain the latest result only since Python cells are analyzed in a way that all previously analyzed cells are analyzed alongside the current one. For more details please see Section 5.3.2 that describes the language context for Python.

For SQL the concept is basically the same as for Python. For each SQL cell, the code is sent to the Databricks SQL service alongside the information about available variable values and default catalog and schema values. Then when the result is returned it is saved to a helper result class that stores all results produced by the SQL service, not only the latest. The reason behind this is that SQL cells are analyzed separately one by one, hence all results need to be stored.

Using this pattern handlers for any language can be added easily and without extensive changes in already existing code.

5.5.2 Analysis context

Based on the design from the previous section we need to design a way to properly store the results of analyzers and also find a way how common information for analyzers can be passed easily. For these purposes, we introduced a so-called *analyzer context*. This class is created in the Dataflow task class which is also a top-level entry point to the analysis. There, common information like output graph and node provider are stored. This analyzer context is then passed through the dispatcher to all called analyzers. When the called analyzer finishes the analysis of a notebook, the results will be stored in the analyzer context under the notebook information and also with a flag that denotes which analyzer produced the result, since we can have a situation that combines multiple approaches together.

When the analysis of all notebooks is finished, all results for all notebooks are taken from the analyzer context and are merged into the final graph.

5.5.3 Worklist algorithm

This section describes the worklist algorithm that should be used in the analysis of notebooks.

Worklist over notebooks

The basic idea of the worklist algorithm is that we will work with the Invocation contexts of notebooks that will contain all data about a notebook, its shared and languages context, and any other data necessary for the analysis. Then we have the Analyzer Context, which will contain data based on the analyzer work (external scanners analyzer mostly) like new invocations that need to be added to the worklist (based on the results from the Python scanner - e.g.: if one notebook calls another one), the results of the analysis for each notebook invocation that has been analyzed at least once, the dependencies of notebooks and the node provider. The Analyzer Context is maintained by the main Dataflow Task class and will be passed to/from analyzers for the analysis of individual notebooks. The worklist algorithm at the beginning creates notebook invocation contexts for each notebook that is available. Then it will add all created invocation contexts to the worklist set. Then the iteration over the worklist ordered set starts. While the set is not empty the algorithm will call the dispatcher for the notebook analysis. The dispatcher would run the external scanner analyzer first (highest priority) and that analyzer would try to analyze the notebook cell by cell. Then when the analysis finishes successfully the analyzer creates results from the data that it got from the external scanner and will compare it with the result we have from the previous analysis (if we have any). If the result changed the notebook invocation context alongside with context of all notebooks the current notebook is dependent on will be added to the Analyzer Context as new invocations that need to be analyzed since analyzers do not have access to the worklist that is located in the Dataflow task class. Hence, the Analyzer Context will be used as a sort of temporary storage of information that needs to be passed from the analyzer to the Dataflow task. Also, the new result from the notebook analysis will be stored in the analyzer context. The External scanner analyzer will then

return true (if the analysis was successful) to the dispatcher and the dispatcher will return true (if successful) to the dataflow task. The dataflow task will take a look at the analyzer context if some invocation context should be added to the worklist. If so, the invocation contexts are added to the worklist and the set in the analyzer context will be cleared. Then the algorithm continues until the worklist is empty. When the worklist is empty, the results for all notebooks are collected and added to the final graph. Once everything is added, the analysis finishes and the process moves on to the next stage.

Figure 5.23 contains a visual representation of the worklist algorithm workflow.

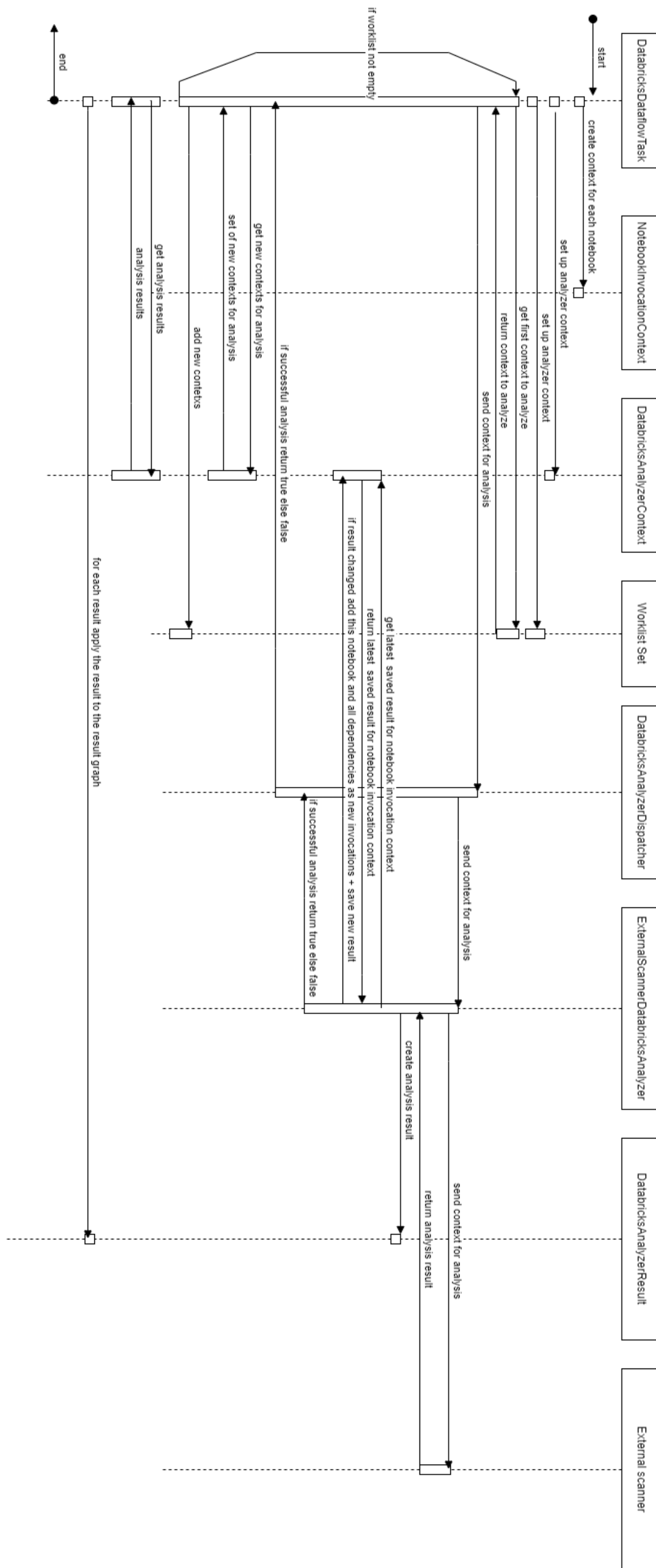


Figure 5.23: Worklist over notebooks workflow

For a better understanding of the algorithm, the following pseudo codes will describe the most important components that contain logic regarding the worklist algorithm.

DatabricksDataflowTask

This section will provide the pseudocode for the top-level part of the analysis algorithm which works with the worklist. This process will happen in the `DatabricksDataflowTask` class.

```
method doExecute(input, outputGraph):
  if no notebooks present in input:
    return

  analysisContext = new context for analysis
  worklist = new worklist instance

  for each notebook on input:
    create an empty invocation context
    add the empty IC to the worklist
  end

  while(worklist not empty):
    notebookIC = worklist.getNextNotebookInvocationContext()
    pass the notebookIC and analysisContext to the dispatcher for
     analysis and wait for results
    add all invocation contexts from analysisContext to the
   worklist
  end

  gather all analysis results from analysisContext and apply them
     to the outputGraph
```

ExternalScannersDatabricksAnalyzer

This section describes the algorithm of the analyzer that works with external scanners.

```
method analyze(analysisContextinvocationContext):
  newResult = new analyzer result for external scnanners analyzer
  notebook = invocationContext.getNotebook()

  for each command in notebook:
    match command type:
      case PYTHON:
        newResult = PythonECS.analyze(command, invocationContext.
   getSharedContext(), invocationContext.getPythonContext())
      case SQL:
        newResult = DatabricksSQLService.analyze(command,
   invocationContext.getSharedContext(), invocationContext.
   getSQLContext())
      default:
        not supported yet

  oldResult = analysisContext.getOldResultforNotebookIC(
   invocationContext)

  if newResult != oldResult:
    add new invocation contexts to analysis context based on
   results from analysis
```

```
18      store new result for invocation context to analysis context
```

Worklist over cells

This section contains an explanation of why we decided that the worklist through the notebook invocation contexts is sufficient.

First, let us describe the idea behind the worklist algorithm through the cells. The worklist algorithm would iterate over the individual cells of a single notebook until the results of the analysis would achieve a consistent state. This would have been needed in case the cells could use something that would be defined in the following cells. Let us demonstrate an example:

```
1  # Databricks notebook source
2  def doSmth(vname, newVal):
3      spark.conf.set(vname, newVal)
4      doCrazy()
5
6  # COMMAND ----------
7
8  doSmth("var1", "valueC")
9
10 # COMMAND ----------
11
12 def doCrazy():
13     print("Hello World")
```

In this example, the first cell written in Python calls method `doCrazy()` which is defined in the third cell. If this behavior would be possible then we would need to have a worklist over cells as well so that we would be able to get the information about the `doCrazy()` method definition to the first cell. However, during our analysis and research, we found out that this behavior is not supported by Databricks. In Databricks the notebooks are executed cell by cell. Each cell can refer to what has been done in previous cells, however, it cannot use anything that will only be defined in the following cells. For illustration, we provide a screenshot of the previous example when we tried to run it in Databricks.

```
Cmd 1

1  def doSmth(vname, newVal):
2      spark.conf.set(vname, newVal)
3      doCrazy()

Command took 0.16 seconds -- by natalia.potocekova@getmanta.com at 3/29/2023, 11:32:52 AM on Natalia Potocekova's Cluster
Cmd 2

1  doSmth("var1", "valueC")

⊞NameError: name 'doCrazy' is not defined
Command took 0.59 seconds -- by natalia.potocekova@getmanta.com at 3/29/2023, 11:32:52 AM on Natalia Potocekova's Cluster
Cmd 3

1  def doCrazy():
2      print("Hello World")

Command skipped
Command took 0.60 seconds -- by natalia.potocekova@getmanta.com at 3/29/2023, 11:32:52 AM on Natalia Potocekova's Cluster
```

Figure 5.24: Error when referencing future definition

As can be seen in Figure 5.24, the notebook execution will end with `NameError` hence this notebook is not valid and cannot be properly executed as a whole.

During our discussions and analysis, we created a few problematic scenarios and tried to analyze if the notebook invocation context worklist is sufficient or if there is a situation that needs the cell worklist algorithm. The following sections will contain examples, a description of the situation, and an explanation of the sufficient solution. The situations will be written in pseudo-code.

Refer to existing spark context variable

In this situation, we have the first cell written in Python that contains a definition of a method that sets a variable with a given name to a spark context with a value `valueA`. Then we have a second cell written also in Python that calls the method that is defined in the first cell with value `var1`. Then we have a cell in SQL that uses the value of the variable called `var1` as a parameter of the query. The last cell is written in Scala and sets the value of the variable called `var1` to a value `valueB`.

In this situation we do not have any issues as the Python scanner will analyze the first cell and then the first two cells. From that, we will get a record in the shared context that notes that has `var1: [valueA]` pair stored. The SQL scanner will get the shared context and will correctly put the `valueA` as a parameter in the query. Lastly, the Scala analysis will add another possible value to var1 so the context will contain the `var1: [valueA, valueB]` record.

Refer to non-existing spark context variable

```
1 % Python cell 1
2 def doSmth(vname)
3   spark.context.set(vname, "valueA")
4 
5 % SQL cell 1
6 SELECT spark.context.get("var1") FROM table
```

```
7
8 % Python cell 2
9 doSmth("var1")
10
11 % Scala cell
12 spark.context.set("var1", "valueB")
```

In this example we have the exact same notebook, however, we switched the second and third cells. So now the SQL cell tries to access a variable that has not yet been initialized in the spark context with any value. There are 2 possible scenarios in this case:

1. Either there is some old record from the previous run of the notebook that has some spark record for the var1 variable with some value and the notebook run will be successful - however in this case our scanner has no chance of knowing the value, hence we would not be able to analyze it either way.

2. The second option is that the var1 does not exist in the spark context at all. In this case, the notebook run fails hence the notebook is not valid.

Hence we don't have to support this situation since it is not a valid state.

SQL function call

```
1 % Python cell 1
2 def doSmth(vname)
3   spark.context.set(vname, "valueA")
4 def doSmth2(vname,newval)
5   spark.context.set(vname, newval)
6   doCrazy()
7
8 % Python cell 2
9 doSmth("var1")
10
11 % SQL cell 1
12 STORED PROC proc1
13   SELECT spark.context.get("var1") FROM table
14 CALL proc1
15
16 % Python cell 3
17 doSmth2("var1", "valueC")
18
19 % SQL cell 2
20 CALL proc1
```

In this case, we have the first cell written in Python that defines two functions that set values for variables in the spark context. The second cell calls the first method that will set the value for `var1` to `valueA`. Then we have an SQL cell that defines a function and calls it (the function uses the value of `var1` in a select statement). Then there is a python cell that sets the value of `var1` to `valueC`. Then there is again an SQL cell that calls the procedure defined in the previous SQL cell.

The callings orders are correct here since the second cell will first create the spark variable `var1` with value `valueA`. Hence the shared context would contain a record `var1: [valueA]`. The shared context would be used by the SQL

cell and the value of `var1` would be correctly found. Then the context would be updated by the python cell to `var1: [valueA, valueC]`. The second SQL cell would work with both possible values in the context, however the correct `valueC` would be there amongst possible values, and the rest would be a result of over-approximation caused by the Python scanner. Hence the situation would be handled properly.

Refer to the method defined later

```
% Python cell 1
def doSmth(vname)
  spark.context.set(vname, "valueA")
def doSmth2(vname,newval)
  spark.context.set(vname, newval)
  doCrazy()

% Python cell 2
doSmth("var1")

% SQL cell 1
STORED PROC proc1
  SELECT spark.context.get("var1") FROM table
CALL proc1

% Python cell 3
doSmth2("var1", "valueC")

% SQL cell 2
CALL proc1

% Scala cell
spark.context.set("var1", "valueB")
spark.context.set("var2", "valueF")

% Python cell 4
def doCrazy()
  file.write("path1", spark.context.get("var2"))

% Scala cell
spark.context.set("var2", "valueG")
```

This situation is similar to the very first example we mentioned in this chapter. Here the first cell defines two methods and the latter one refers to a method that is defined in Python cell 4 which means that it is defined later in the notebook. That would not be a problem if the method `doSmth2(vname,newval)` has not been called before the definition of the `doCrazy()` method. Hence this notebook would end in `NameError` since the method `doCrazy()` would not have been known when calling the `doSmth2` method.

Refer to a value of another notebook

```
% Python cell 1
tableName = dbutils.notebook.run("notebookB", {catalog: "c",
    schema: "s"})

% Python cell 2
doSmth(tableName)
```

In this case, the first Python cell contains the calling of another notebook called notebookB and uses its return value as a parameter for a method that will do something with it. In this case, when analyzing the first cell, we will find out that currently analyzed notebook calls notebookB and that we have no record in context for its return value. Then it would be recorded that the notebookB has been called (and hence needs to be analyzed) and the analysis would continue with an unknown return value. Then the analysis of the current notebook would end after the second cell which worked with the UNKNOWN value and both notebookB and the current notebook would be added to the worklist again. Hence the problem of not having a value from the previous cell would be solved by the notebook invocation context worklist algorithm.

After analysis of all of the mentioned examples, we concluded that the worklist over the notebook invocation contexts would be sufficient for now.

5.6 Result graph

The last thing that needs to be properly discussed and analyzed is the result graph that is presented to users in the Manta viewer. In this section, we describe what kinds of nodes we have to represent Databricks entities in Manta and their hierarchy.

5.6.1 Data entities

Firstly we take a look at the data entities. In Databricks, we have a three-level namespace that is represented by catalogs, schemas, and tables. It is only natural that for easy and useful representation we use the three-level hierarchy in our graph as well. Hence, all tables are represented by a top-level node that is named using the catalog name, a second-level node that is named using the schema name, a third-level node that is named using the table name, and then on the fourth level, we visualize the columns of the table. In case we know nothing about the columns of the table there will be one column node present with the name TABLE COLUMNS UNKNOWN. In the details of the column, we can show the details we obtained (if any) from the Unity Catalog API.

Also, since all data entities are bound to the instance they are created in, the whole hierarchy described above has to be nested in a node that represents the current instance.

Figure 5.25 shows the mentioned hierarchy of nodes for representing tables. There is an instance called databricksDemo which contains two catalogs named quickstart_catalog and system. The quickstart_catalog contains schema called lineagedemo and inside of it is one table called grades. Since we have no information about columns, there is only one column with the TABLE COLUMNS UNKNOWN name. The system catalog contains a schema named information_schema, inside of it table named tables that has one column named table_name.

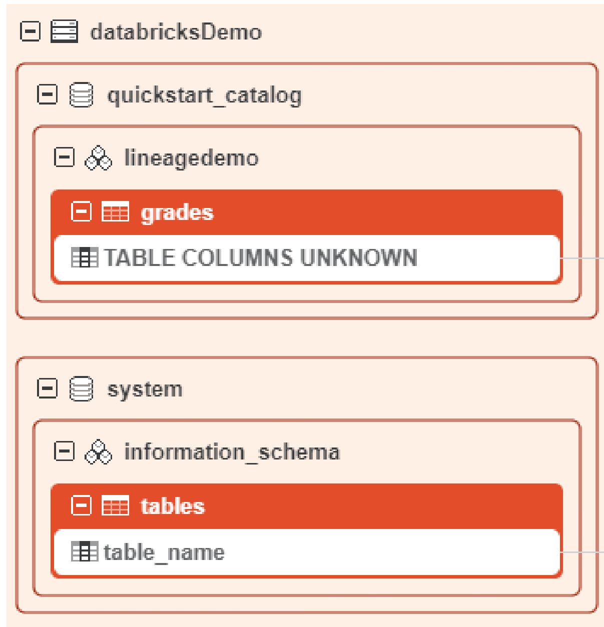


Figure 5.25: Data entities node hierarchy example

5.6.2 Source codes

Other entities that have to be visualized properly are the source codes. In this thesis, we focus on the notebooks. In Databricks workspace, notebooks can be nested in different folders. We would like to preserve this hierarchy in our graph as well since users have to be able to distinguish between two notebooks that have the same name but are stored in different folders. Hence, there should be nodes that represent the folder structure where the notebook is saved. Then there is a node that represents the notebook itself. Under this node, the results of the analysis are attached. Figure 5.6.2 shows an example of a notebook called `Python test` that is stored in a folder `test_folder`. Under the `Python test` notebook node there is an attached result from the Python scanner that shows a read from the standard input.

In the future, queries and jobs will be represented in a similar way to notebooks.

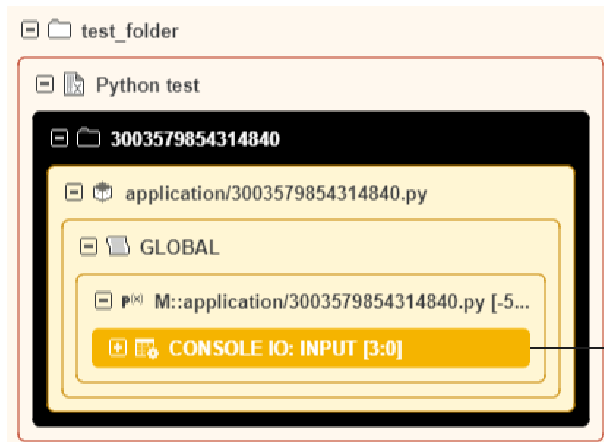


Figure 5.26: Data entities node hierarchy example

6. Implementation

Now that we have described our solutions for the biggest issues and obstacles we encountered, let us take a look at the way these solutions are implemented. The following sections focus on specific parts of the implementation such as extraction of information, context implementation, worklist algorithm, and extensions in the Python scanner.

6.1 Extractor

Extraction is usually the first step when trying to visualize lineage, hence it is only natural to dive into it as first. In the following sections, we describe the common model, representing Databricks entities, that is used in the scanner and then we describe the Hive metastore extraction in more detail as it is the subject of this thesis. Databricks scanner contains also classes that support the Unity Catalog extraction, however, since they are not a part of this thesis project they were developed by other Manta employees and hence will not be described further.

6.1.1 Common model

Since there are multiple ways of extracting information from Databricks (Unity Catalog or Hive metastore extraction) we need to have a unified way to represent them. That is why we created a common set of interfaces for each entity we need to extract. The list of the entities is as follows:

- catalog
- column
- column lineage
- notebook
- notebook command
- query
- schema
- table
- table lineage
- workspace

There were also a few more entities we needed to add artificially for easier work with the extracted data. The first step was creating a column identifier which is basically just a connection of the column to its table. Another connection we needed to add was the table identifier. This identifier connects the table with its schema and catalog.

The next step we needed to do was to add a proper way of storing the extracted data. To do so, Databricks Extractor uses serialization. Hence, we defined another set of interfaces that define a way to serialize all information we want to pass to the analysis. From the list of entities above, the only entities that do not have the serializable interface are the catalog, schema, and workspace. This is due to the fact that catalog and schema are a part of the table identifier and the workspace is a part of the notebook information.

Of course, alongside the interfaces, we need to have a proper implementation of them. Each of the extracted entities has its own class that implements a given interface. The naming convention goes as follows: `ExtractedDatabricks<name_of_the_entity>`.

6.1.2 Hive metastore

Now that we know the common model used in the scanner, we can describe in more detail the classes used for the Hive metastore extraction.

DatabricksMetastoreConnectionManager is a class responsible for managing the connection to Databricks through the JDBC connector we mentioned in Section 4.5. The two basic actions this class does are creating a new connection and verifying that the connection is still open.

DatabricksMetastoreClient represents the client that is used for communication with the Hive metastore. In this class, there are methods for obtaining all of the information available in Hive metastore such as schemas, tables, table columns, and views. This class uses the connection manager described above to connect to the Hive metastore.

DatabricksMetastoreDataProvider represents the top-level class responsible for obtaining the information from the Hive metastore. The provider uses the client class to obtain the available things from the metastore. For all information that cannot be obtained returns either an empty set or other suitable representation of empty information.

TableInfo is used to represent the table and view raw information obtained from the JDBC driver connected to the Databricks instance.

MetastoreTable represents the class that takes the raw information and bundles them together properly. This information is then transformed into the common model described above.

6.2 Dataflow Generator

In this section, we describe our implementations of the most important or interesting classes that can be found in the Dataflow Generator part of our scanner. We describe the implementations of the contexts we designed in Section 5.3, the worklist algorithm-related classes from Section 5.5.3

6.2.1 Context

In this section, we describe how we implemented the shared and language contexts we designed in Section 5.3. We show the shared context implementation as first and then the language context implementation for both Python and SQL.

Shared Context

As mentioned in Section 5.3.1 the shared context is responsible for storing the values that are put in the spark context for the Databricks notebook. These values can then be used by any language the cells of the notebook are written in. This section will describe what kinds of information can be stored in the shared context implementation for now and what are the plans for the future.

For now, we store only variable names and their values in the shared context. Hence the context works with classes that represent the variable value. As of now, we know, that Databricks supports only simple types like string, number, or boolean as a value for global variables stored in the spark context. Due to this reason, we decided to create a class named `VariableValue` which is an abstract parent for each class that should represent any value of a variable. Then we created classes for each allowed type of value. Figure 6.1 shows the hierarchy of the values and methods they provide.

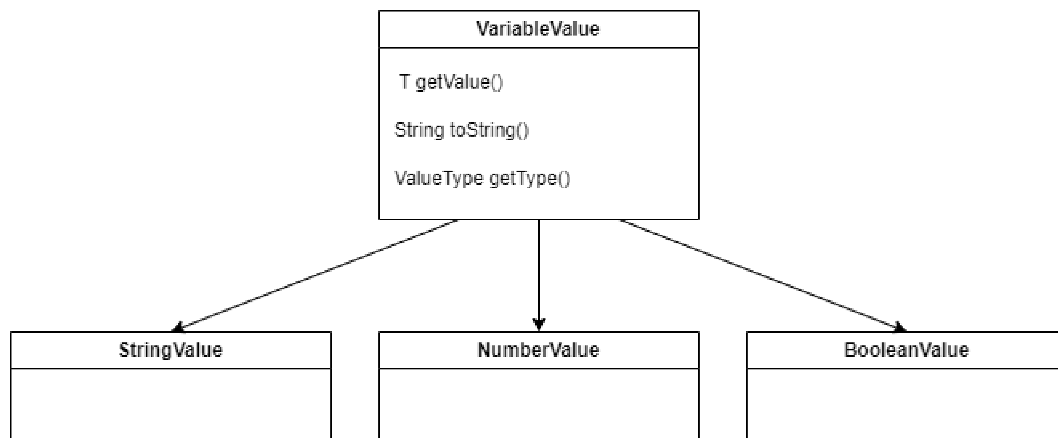


Figure 6.1: Hierarchy of variable values classes in shared context

As can be easily deduced from the names, the `StringValue` class represents the string values, the `NumberValue` represents any kind of numerical value (integer, float, etc.), and the `BooleanValue` represents the boolean values of variables.

Should any other variable value type be needed, it can be easily added. The only thing that has to be done is to implement the `VariableValue` class with a defined generic type T (type of value) and that's it. All methods defined in the context work with the `VariableValue` itself, so no additional changes in the context should be needed.

In the future, it is planned to also support creating tables and temporary views so the model of classes used in shared context will be enriched to support these features.

The interface provides methods for the following actions:

- get a value for a variable specified by its name

- add a value for a variable specified by its name
- get all valid combinations of variable values in the context
- create a copy of the context

All operations with the shared context use this interface and not concrete implementations of it. Thanks to this, It does not matter what kind of implementation (Cartesian product, prototype-like, etc.) we choose as long as we properly implement the interface. In our case, we implemented the version of shared context that uses the Cartesian product to return the valid variable combinations. This implementation can be found in the class `SharedContextCartesianImpl`.

In order to represent one combination of possible variables values combination we implemented the `VariableValuesCombination` class. This class is a wrapper class for a simple map that has names of variables as keys and `VariableValue` as value. We decided to provide this class for easier manipulation with the combinations. Also, actions that would be frequently used by anyone working with the context can be added easily to this class. Hence, the methods are implemented only in one place and other scanners can only use them, there is no need for them to have their own implementation.

Language context

As mentioned in Section 5.3.2 the language-specific contexts are used to store any additional information that can be used during the analysis of a code in a given language or any result information specific for that language that should be used later in the analysis. In this section, we describe the interface and classes used in the parent interface of all language-specific contexts, and in the following subsections, we describe concrete interfaces for Python and Databricks SQL languages.

What all language-specific contexts have in common is that during the analysis of a cell, some error could have occurred that we would like to inform users about. Due to this reason, we have `AnalysisErrorRecord` class that contains the error type and description. Another thing that may be useful for any language is the option of adding additional resources available for the analysis (like libraries). Due to that, we added the `Resource` class that contains the resource name and source code.

The interface provides methods for the following actions:

- get records for errors specified by their type
- save a new error record - given the type and description of the error
- get a resource (library or any other kind) by the name
- get all resources stored in the context
- save a new resource - using the resource name and code
- create a copy of the context

The implementation of this interface can be found in the abstract class called `ALanguageSpecificContextImpl`. This class serves as a parent class for all language context implementations. Operations with resources and errors are already implemented in this class. Hence when adding a new language context, only the language-specific functionality has to be implemented.

Python language specific context

Additionally to the features mentioned in the previous section, Python language context has to store the previously analyzed cells (this is instead of huge Python scanner analysis context classes so that we would not lose information about previously available libraries, variable values and any kind of information that could be accessed in the current cell). Hence, we created `AnalyzedCell` class that contains the source code of a cell that has already been analyzed. There is an interface `PythonLanguageSpecificContext` that extends the `LanguageSpecificContext` interface to define all methods necessary for the Python language context. An implementation of this interface can be found in the

`PythonLanguageSpecificContextImpl` class. The interface adds the following methods:

- add the cell that has already been analyzed to the context
- get all previously analyzed cells

Databricks SQL language-specific context

Additionally to the common properties, the Databricks SQL requires to remember the default catalog and default schema value. However, since these values have to be string values, we have decided to require the string values only.

Again, there is an interface called `DatabricksSQLLanguageSpecificContext` that extends the `LanguageSpecificContext` interface to define all methods necessary for the Databricks SQL language context. An implementation of this interface can be found in the `DatabricksSQLLanguageSpecificContextImpl` class. The interface adds the following methods:

- get the values of the default catalog
- add a value of the default catalog
- get the values of the default schema
- add a value for the default schema

6.2.2 Analysis

In this section, we focus on classes that are used for the notebook code analysis. Firstly we take a look at the main analysis class called `DatabricksDataflowTask`, then we describe the classes used for the worklist algorithm and last but not least we describe the analyzer for external scanners.

Dataflow Task

As mentioned above, the `DatabricksDataflowTask` is the main class for the analysis. This class is responsible for reading the extracted information, performing the analysis of the notebooks using the worklist algorithm, and creating the final graph using the results of the notebook analysis. Let us now describe the classes that are used for each of these responsibilities.

DatabricksGraph represents the input for the analysis. This class contains the notebooks and queries available from extraction. This class has been developed in cooperation with Manta employees as queries were not the subject of this thesis.

DatabricksNodeProvider is a class that provides nodes that represent notebooks or tables. In case the nodes already exist, the existing instances are returned and in case there is no suitable node a new one is created. This class has been developed in cooperation with Manta employees since it also supports Unity Catalog features that are not a part of this thesis.

DatabricksAnalyzerContext represents the context that is passed to the analyzers. This class contains the final `Graph` instance that represents the output graph, also the `DatabricksNodeProvider` is present there so that proper nodes can be found in analyzers. Another important thing that is stored here are the actual notebook summaries for given invocation contexts (necessary for the worklist algorithm to work correctly), the list of new invocation contexts that should be added back to the worklist set, and the pairing between notebooks and their `DatabricksNotebookGraph` information (lineage information). This class is used by both the Unity Catalog analyzer and External scanners analyzer.

Once all data are properly read, the worklist algorithm takes place. The following section describes the classes related to the worklist algorithm.

Worklist

As we described in Section 5.5.3 we use the worklist algorithm to ensure we get the correct results back from the analysis of notebooks. Let us now describe the classes used in the algorithm implementation.

DatabricksWorklist is a class that represents the worklist. The class contains a queue for invocation contexts of notebooks and provides methods like adding and removing invocation contexts into the queue.

NotebookInvocationContext is an immutable class representing an invocation context of a notebook. Invocation context is a basic unit over which the worklist algorithm is executed. It contains all information that is necessary to track in order to determine if the results are stabilized or not. To be precise the invocation context contains the shared context reflecting the initial state at the beginning of the analysis, language contexts reflecting the initial state at the beginning of the analysis, notebook graph (contains notebook cells, name (identifier)).

NotebookSummary is a class for storing the notebook and its shared and language contexts that are updated throughout the analysis of a notebook. In case the current invocation context and notebook summary differ from the invocation context and notebook summary from the previous run of analysis, the updated version of the invocation context is created and put back to the worklist so that it would be analyzed again.

Analyzers

Analyzers are responsible for applying a certain analysis approach to get lineage results from notebooks. The following classes are used to analyze notebooks.

DatabricksAnalyzerDispatcher is the top-level class responsible for applying the analyzers in the proper order based on their priority. There is also an option to define which analyzer should be used. This can be specified via enum **DatabricksAnalyzerDispatcherMode**.

DatabricksAnalyzer is a common interface for every analyzer used in the Databricks scanner. Currently, it only contains one method called `analyze`. This method takes the **DatabricksAnalyzerContext** and **NotebookSummary** as parameters. It returns a boolean value representing if the analysis was successful or not.

As we mentioned above, we currently have two analyzers in our scanner - one for Unity Catalog and one that uses external scanners from Manta to analyze notebook source codes. Since the Unity Catalog part of the scanner is not part of this thesis, we will only describe the latter analyzer.

ExternalScannersDatabricksAnalyzer represents the approach of analyzing notebook source codes. This class is responsible for calling the Python scanner and Databricks SQL scanner. This is done via handlers. The **PythonCommandAnalysisHandler** class is responsible for using the ECS for Python to analyze the Python commands. There are several steps that have to be done in order to do so:

1. Create an empty `DatabricksInsighter` instance.
2. Create a `DatabricksOutsight` instance filled with values from the shared context.
3. Create a `PythonDatabricksConfiguration` instance using the created insighter and outsight.
4. Add additional libraries (if there are any present) to the configuration.
5. Create the final version of the code that should be sent to analysis - use all previously analyzed cells and put them before the current cell code. Also, add initialization of the spark context at the beginning so that the Python scanner would be able to work with the spark variable.

6. Use ECS to get the result of the analysis. Pass the configuration, script name, node representing the notebook and the code to be analyzed as parameters.

7. In case the analysis was not successful return false

8. Save the result as the `PythonExternalScannerResult` in case the analysis was successful.

9. Create an Insight from the Insighter and based on the values in it update the shared context.

10. Update the language context by adding the current cell to the list of already analyzed cells.

11. Return true.

PythonExternalScannerResult is a wrapper class that connects notebook with the result from the Embedded code service for Python. This result is then used when creating the final graph that should be displayed to users. This is done via a method `apply` from the common interface for results **DatabricksAnalyzerResult**. The `apply` method for Python takes the result from the ECS, and checks if it has any lineage. If it does, firstly the pin nodes are properly connected under one node that represents the spark context. Then, the result is merged to the final graph.

Now that we described the process for Python language, let us do the same for Databricks SQL language.

DatabricksSqlCommandAnalysisHandler class is responsible for using the ECS for Databricks SQL to analyze the SQL commands. In order to do so, these steps need to be executed:

1. Iterate over all possible values of the default catalog stored in the language context.

2. Iterate over all possible values of the default schema stored in the language context.

3. Get all possible variable values combinations from the shared context.

4. Create new `ContextDatabricksSQLEmbeddedCodeConfiguration` for the ECS using the catalog value, schema value and variable values.

5. Use the ECS to analyze the current SQL cell.

6. Save the result from ECS as the `DatabricksSQLEmbeddedCodeResult` and based on it update the language context accordingly.

7. Create a new `DatabricksSqlExternalScannerResult` instance when the final result should be returned using the embedded code results saved from previous analysis.

DatabricksSqlExternalScannerResult is a wrapper class that stores results from the analysis of each of the SQL cells present in a notebook. Then same as for Python, once the final graph needs to be created, the `apply` method from the common `DatabricksAnalyzerResult` interface. This method iterates over each of the results and merges them into the final graph.

6.3 Python scanner

Since Python scanner has already been developed, the only thing we had to do was add new propagation modes to support Databricks spark methods for working with variables, implement the Insighter, Insight and Outsight classes so that the Databricks scanner can properly pass information into the Python scanner and also get additional information out of the Python scanner. We also needed to extend the Embedded code service for Python to support Databricks. The following subsections describe each extension in more detail.

6.3.1 Python scanner classes

For proper communication between Python scanner and Databricks scanner when it comes to shared variables, we had to implement the following classes (can be found in the Python scanner analysis module in the `externalinfo.databricks` folder):

DatabricksInsighter is an implementation of the Insighter interface specifically for Databricks. In the insighter, we had to implement the method `createInsight` that takes the stored information and transforms them into an immutable object that is used later in the Databricks scanner. To do so, we created a few helper classes for easier representation of variables and their values.

DatabricksVariableValueInfo is a wrapper class that contains all information about variables we need to have in the shared context in Databricks. To be precise the class contains the variable value, type, and validity information. The validity information is an enum that is used to determine how precise the value is. It can be either so-called constant flow (Python scanner is 100% sure about the value) or it can be value flow which represents variable values that were somehow processed (e.g.: string concatenation) or their value is unknown from the start (e.g.: value read from the standard input).

DatabricksInsight is an immutable object that is used in Databricks scanner to properly update the shared context values. The insight contains methods for returning all of the variables that are available and returning information about a specific variable based on the variable name.

DatabricksOutsight represents the object that is passed into the Python scanner before the analysis is filled with values of variables that were already present in the shared context. Then, whenever a value from shared context is requested in code, the Outsight is used to find proper values. From the Python scanner

point of view, the Outsight is read-only, from the Databricks scanner point of view the Outsight is write-only.

6.3.2 Propagation modes

In order to be able to work with the variables from shared context properly, we had to add two new propagation modes to the Python scanner's pyspark plugin.

SparkConfGetPropagationMode is the propagation mode responsible for the handling of the `spark.conf.get` method used for getting variables from the spark context. This propagation mode uses the `DatabricksOutsight` class to create flows that represent the variable values properly. The `spark.conf.get` method takes one parameter that represents the variable name for which the value should be taken out of the spark context. The handling of this method in our propagation mode goes as follows:

1. Take all `LiteralFlows` that contain string value and represent variable name from the propagation source.
2. For each possible variable name take all possible variable values from the Outsight based on the variable name (if present) and create proper flow based on the validity info of the variable value.
3. Register all newly created flows to the propagation target. Also, create `InputPinFlow` that will be used to create an input pin node for the variable and register it to the propagation target.

SparkConfSetPropagationMode is the propagation mode used for handling the `spark.conf.set` method used for storing variables into the shared context. The `spark.conf.set` takes two parameters - variable name and variable value that should be stored in the spark context. This propagation mode uses the `DatabricksInsighter` to store values of shared variables so that Databricks scanner would have all necessary information available. To handle this method the following steps need to be done:

1. Take all `LiteralFlows` that contain string value and represent variable name from the propagation source's first argument.
2. Take all `LiteralFlows` that represent variable value from the propagation source's second argument.
3. For each possible variable name and for each of the possible variable values add a new record to the Insighter.
4. Take all `PythonFlows` present in the second argument values in the propagation source. These flows represent the unknown variable values as there can be read from the standard input, read from a file, database, or any other action of a similar kind.
5. For each possible variable name and each of the unknown value flows create a record in the Insighter with the value set to `UNKNOWN`. Create the `OutputPinTerminal` that will be used to create output pin for a given variable and register it to the propagation target.

6.3.3 Embedded code service

As we mentioned in Section 2.3 the Embedded code service for Python is used as a sort of bridge between Python scanner and other technologies that have Python embedded in their source codes. Since the Embedded code service for Python has already been implemented, the only thing that had to be done was extending the ECS to support Databricks. In order to do so, the following classes were implemented:

PythonDatabricksConfiguration represents the configuration specific for the Databricks technology. The configuration contains the insighter and outsight classes that are supposed to be passed to the Python scanner, the name of the notebook that is supposed to be analyzed, and external libraries.

DatabricksOrchestrator is the class responsible for properly preparing the input on the file system for the Python scanner. In our case, the orchestration process goes like this:

1. A new file with the name of the notebook (specified in the configuration) is created in the input folder.
2. The content of the notebook is copied from the input to the file on the file system.
3. For each available resource a new file in the input folder is created and filled with the library content.
4. An initial `EntryPointLocation` is created based on the created files in the input folder and returned.

We also had to add the Databricks technology to the main `Orchestration` class so that the proper orchestrator would be called.

6.4 Testing

To test the implemented code and scanner as a whole, we used the three-level testing approach.

The first level are the **Unit tests**. Each class that was implemented has a set of unit tests for all accessible methods that test the correct input and output as well as the behavior in the edge cases such as null values.

The second level are the **Integration tests**. These tests are implemented for the extractor and dataflow generator separately. The extractor part of the scanner is tested if all of the test data from our test instance are properly extracted into a temporary folder. For the dataflow generator, the results of the analysis are tested. That means that we prepare the input for the analysis, run the analysis and check if the result graph contains all edges and nodes we expect it to contain.

The third and final level is the **Qualification testing**. In these tests, we test the whole scanner if it works as expected in the Manta Flow platform. These tests are performed manually on the installed Manta instance. The test data are extracted from our test instance using the `ExtractionScenario` and then the

analysis is run using the `DataflowScenario`. In case both scenarios are finished successfully the result graph is viewed in the Manta Viewer and compared with the expected results.

7. Evaluation

In this chapter, we would like to demonstrate that the main features implemented in this thesis work as expected, namely the worklist algorithm, the shared and language contexts, supporting analysis of both Databricks SQL and Python language cells, and usage of Pin nodes in Python scripts. Since the worklist algorithm is the backbone of the analysis, we cannot demonstrate it on its own, however, its correctness is a necessary pre-requisite for the correct results of all other features.

For showing the results of the mentioned features we use the script shown in Figure 7.1.

Figure 7.1: Databricks notebook used for evaluation

Note that the analyzed source code was created to highlight the scanner functionality and it may not make sense to execute it in a production environment. However, with some minor adjustments, the example would make sense even to

be used by users in the real world. In terms of the Python and SQL syntax, all source code is valid.

7.1 Databricks SQL and Python interaction example

The first major goal of this thesis was to support analysis for both Python and Databricks SQL languages. In this section, we show that we fulfilled this goal and we show that we managed to support the interaction between the mentioned languages.

This feature is related to cells `Cmd 3` to `Cmd 5` in Figure 7.1.

The `Cmd 3` cell (written in Python) defines a simple function, that sets three spark context variables called `source.tableName`, `source.name_col` and `source.age_col`. That means, that the values that were passed as parameters of the function are stored in the spark context and hence are available to all languages in the notebook.

The `Cmd 4` (written in Python) cell calls the function defined in `Cmd 3` cell and passes the following values as arguments:

- a value read from a standard input using the `input()` function for the table name
- an array of constants (`"name"` and `"age"`) for the column names

Then the `Cmd 5` cell (written in SQL) executes a simple `SELECT` query on the table with the name of the table and columns taken from the spark context.

The result graph for this part of the notebook can be seen in the blue rectangle in Figure 7.2. The light red node denotes a Databricks table and the dark red node represents the result of the `SELECT` query from the `Cmd 5`. The name of the Databricks table is `unknown`. This proves, that the interaction between languages (and shared context implementation alongside it) works as expected. Let us explain why. Firstly Python scanner analyzes the part where the table name is the value that was read from the standard input. Since in static analysis the value is not known, the Python scanner sets the value to `unknown`. Then when Databricks SQL takes the name from the shared context only the name `unknown` is available. In this example, we can also see that the language context implementation works properly as well since we did not have any information about the catalog and schema names for the table, but the Databricks scanner used the default values set in the Databricks SQL language context which are set to the value `default`.

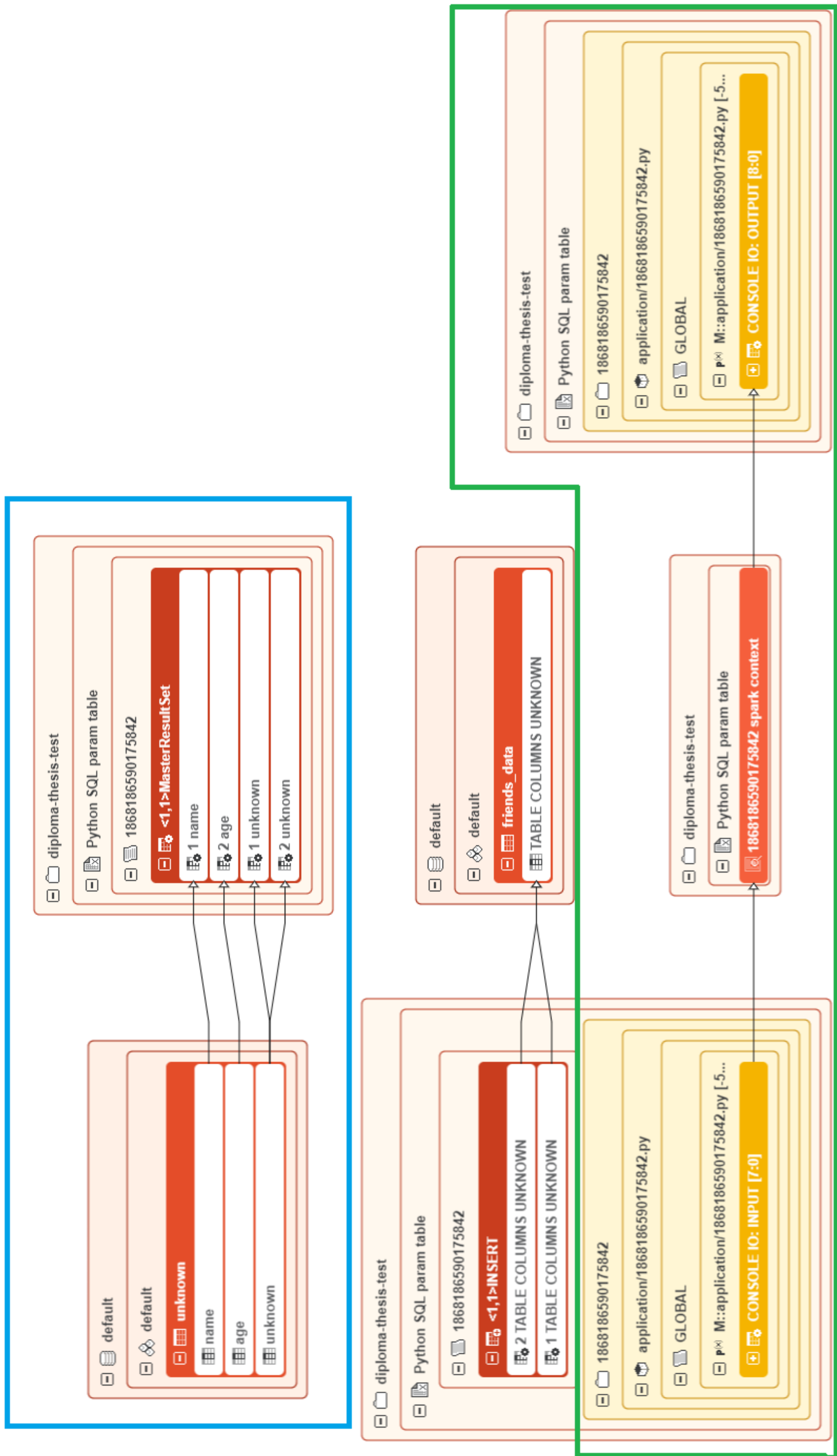


Figure 7.2: Result graph for the evaluation example

7.2 Python script with Pin nodes example

Another important feature we would like to demonstrate is using the Pin nodes to properly connect the shared context usages in the Python scanner. This feature is related to cells `Cmd 4` and `Cmd 6` in Figure 7.1.

As we already mentioned in the previous section, the `Cmd 4` cell calls the function that stores the shared context variables. However, for this feature, the important part is, that the table name is read from the standard input via the `input()` method.

Then the `Cmd 6` cell takes the value of the table name variable from the shared context and prints it to output using the `print()` method.

The result graph for this part can be seen in the green polygon in Figure 7.2. There are two yellow nodes connected through a light red node. The first yellow node is a Python node representing the reading from standard input. Then the edge to the light red node represents writing into the spark context. The light red node represents the spark context itself. Then the last yellow node represents write to the standard output produced by the Python scanner. The edge from the spark context node to the standard output write node represents reading from the spark context. You may have noticed, that there is no edge that would connect the spark context and the Databricks SQL result in the previous example. This is caused by the fact, that Databricks SQL does not support pin nodes yet, hence there is no way to connect it.

7.3 Limitations and Future Work

As we demonstrated in previous sections, our Databricks scanner implementation provides a solid base for the analysis of simple Databricks notebooks written in Python and SQL. However, there are features that need to be implemented in the future to analyze the Databricks notebooks fully. In the following sections, we describe the limitations of the current implementation and our plans for the future.

7.3.1 Notebooks calling each other

As it turned out, Databricks supports too many concepts and features and to support it all would be too large for this thesis. Hence we decided to create a scanner that supports the base concepts and would be easily extendable with new features. One of these features is the ability of notebooks to call each other. To add support for this feature, the language and shared contexts would have to be extended to store the notebook results and to note that a certain notebook called another notebook with given parameters. The results would then have to be taken into account when deciding if the notebook should be put back to the worklist. Also, a mechanism for exchanging notebook parameters would have to be designed. Last but not least few propagation modes would have to be added to the Python scanner in order to support the `dbutils.notebook.run` method responsible for calling other notebooks.

7.3.2 Latest result from Databricks SQL

Another interesting feature, that Databricks added shortly before the implementation of this thesis has finished, is the ability of Python, Scala, and R to access the latest SQL result by using the `_sqldf` dataframe. Using this feature, the latest query results produced by SQL cells in the notebooks can be directly represented as dataframes and further processed by the mentioned three programming languages. To add support for this feature a mapping between the results produced by Databricks SQL scanner and Python scanner inputs would have to be designed and implemented. After this only small changes in the Python scanner would be required like adding proper propagation modes.

Temporary views and tables in Spark

Regarding the spark context support, in our thesis, we focused only on variables and their values. However, the spark context also provides an option for sharing the dataframes. Hence another feature that has to be implemented in the future is extending the shared context to store the dataframe values similarly to variables store their values. It is probable that the model used in context would have to be extended with classes that would represent columns or table rows. Alongside that, proper propagation modes would have to be added to the Python scanner.

Adding support for Scala and R

As we mentioned before, in our thesis we support only Python and SQL languages as currently, Manta does not support the Scala and R languages. However, if in the future the scanners for these languages are available, the support for the languages could be added to the Databricks scanner as well. To do so, a few steps have to be done. The first step is creating ECS for each language and the second step is extending the external scanner analyzer to call respective ECS, store its results, and define how the results should be merged into the final graph. After these steps, the support for Scala and R should be finished from the Databricks scanner point of view.

7.4 Lessons learned

In this section, we would like to summarize the main lessons we learned during our work on this project. However, we will mention only the main points we consider useful for someone who would like to create a similar scanner in the future.

The first surprising and interesting discovery we made was that constants can be useful for data lineage. In Manta the constants were mostly ignored when producing lineage. For example, writing a constant value to standard output in Python using the `print()` function would result in an empty lineage. This proved to be a big obstacle in our case since constants are used a lot in Databricks to denote tables name, schemas, catalogs, queries, and other entities. Also, the shared variable names and values can be constants but have to be tracked in order to compute a correct lineage for a Databricks notebook. Hence we needed

to come up with different solutions on how to deal with this issue - for example using the Insight and Outsight for passing constants etc.

Another interesting issue related to the constant was connecting scanners with completely different workflows and analysis mentality. As we mentioned above, the Python scanner mostly ignores constants that are not important for the lineage, however, in order to be able to analyze a SQL script by the Databricks SQL scanner, the constant values for the table names had to be provided, otherwise, the SQL scanner would fail. We solved this issue by using the language context and the shared context in combination with Python Insight and Outsight. Python Insight and Outsight were used to track constants and then the contexts were used to pass these values to the Databricks SQL scanner.

The last but definitely not least discovery we would like to mention is the fact that Databricks does a lot of processes and editing in the background. In order to be able to compute the lineage properly, we had to dive deep into the Databricks processes (which was hard in some cases as there has not been a lot of documentation on them) and simulate them in our scanner. An example of this could be the initialization of a spark context for a notebook. Databricks does this in the background automatically, so the notebooks can use the `spark` variable without creating it first. However, the Python scanner needs to know that the `spark` variable is representing a spark context, hence we had to manually add the initialization ourselves.

To summarize, creating this scanner was a new challenge that has never been done in Manta before. The project discovered some major issues and challenges that had to be overcome in order to compute a lineage properly. On the other hand, this project also helped us to shape our view and perspective on what is important for lineage and what is not. Connecting different technologies together has proven to be a tough challenge, however, the outcomes are really promising, and in the future, hopefully, more scanners like this can be created.

8. Conclusion

In this thesis project, we have managed to develop a new Databricks scanner capable of analyzing simple notebooks written in Python and Databricks SQL. The scanner is a part of the Manta Flow production deployment in the preview mode.

The scanner currently supports the elementary functionality of Databricks notebooks written in Python and Databricks SQL, such as extraction of notebooks and data entities from Unity Catalog and Hive metastore, and interaction of languages through the shared context. However, to be able to fully analyze the Databricks notebooks, several features need to be implemented in the future. A few examples are notebooks calling each other, using the latest SQL result, or temporary views and tables created in Python that can be used in SQL.

As we have shown in the previous chapter, the scanner works as expected and provides a reasonable graph for this project's main goals.

We also managed to analyze the workflow of notebooks calling each other, however, since the scope of the task would be too large it would result in extending the scope of the thesis way beyond what was initially intended. However, when implementing this feature in the future, the workflow analysis and design proposed in this thesis can be used as a starting point which could make the development easier.

Additionally, the design of the external scanners analyzer provides an easy way of extending the Databricks functionality when it comes to the notebook cell languages. Once Manta has the Scala and R scanners available, including them in the Databricks scanner notebook analysis should not be hard.

Bibliography

[1] URL: https://aws.amazon.com/.

[2] URL: https://azure.microsoft.com/en-us.

[3] URL: https://cloud.google.com/.

[4] *_sqldf not defined.* URL: https://community.databricks.com/s/question/0D58Y00008tImnwSAC/sqldf-not-defined.

[5] *ACID transactions.* URL: https://www.databricks.com/glossary/acid-transactions.

[6] *Can you share variables defined in a Python based cell with Scala cells?* URL: https://community.databricks.com/s/question/0D53f00001GHVQBCA5/can-you-share-variables-defined-in-a-python-based-cell-with-scala-cells.

[7] *Configure the Databricks ODBC and JDBC drivers.* URL: https://docs.databricks.com/integrations/bi/jdbc-odbc-bi.html#jdbc-driver.

[8] *Data lake.* URL: https://en.wikipedia.org/wiki/Data_lake.

[9] *Data lineage.* URL: https://www.techopedia.com/definition/28040/data-lineage.

[10] *Data warehouse.* URL: https://en.wikipedia.org/wiki/Data_warehouse.

[11] *Databricks.* URL: https://www.databricks.com/.

[12] *Databricks JDBC Driver.* URL: https://mvnrepository.com/artifact/com.databricks/databricks-jdbc/2.6.25.

[13] *Databricks widgets.* URL: https://docs.databricks.com/notebooks/widgets.html.

[14] *Develop code in Databricks notebooks.* URL: https://docs.databricks.com/notebooks/notebooks-use.html#explore-sql-cell-results-in-python-notebooks-natively-using-python.

[15] *Get a list of queries.* URL: https://docs.databricks.com/api/gcp/workspace/queries/list.

[16] *Hive foreign keys?* URL: https://stackoverflow.com/questions/9696369/hive-foreign-keys.

[17] *How to work with multiple languages on Databricks.* URL: https://medium.com/@robin.loche/how-to-work-with-multiple-languages-on-databricks-e22dea9f8c7.

[18] *i want to use simba.spark.jdbc driver in sprint boot to connect to databricks with token.* URL: https://stackoverflow.com/questions/71291919/i-want-to-use-simba-spark-jdbc-driver-in-sprint-boot-to-connect-to-databricks-wi.

[19] *Interact with external data on Databricks.* URL: https://docs.databricks.com/external-data/index.html.

[20] *JayDeBeApi 1.2.3.* URL: https://pypi.org/project/JayDeBeApi/.

[21] *Jobs.* URL: https://docs.microsoft.com/en-us/azure/databricks/dev-tools/api/latest/jobs.

[22] *Jupyter Notebook.* URL: https://www.databricks.com/glossary/jupyter-notebook.

[23] *List catalogs.* URL: https://docs.databricks.com/api/gcp/workspace/catalogs/list.

[24] *List metastores.* URL: https://docs.databricks.com/api/gcp/workspace/metastores/list.

[25] *List schemas.* URL: https://docs.databricks.com/api/gcp/workspace/schemas/list.

[26] *List tables.* URL: https://docs.databricks.com/api/gcp/workspace/tables/list.

[27] MANTA Python scanner. *Team software project, MFF UK, 2021.*

[28] *Unified engine for large-scale data analytics.* URL: https://spark.apache.org/.

[29] *What is Delta Lake?* URL: https://docs.databricks.com/delta/index.html.

[30] *What is delta-lake?* URL: https://www.databricks.com/blog/2020/01/30/what-is-a-data-lakehouse.html.

[31] *What is Unity Catalog?* URL: https://docs.databricks.com/data-governance/unity-catalog/index.html.

[32] *What the heck is a lateral join anyway?* URL: https://jonmce.medium.com/what-the-heck-is-a-lateral-join-anyway-4c3345b94a63.

List of Figures

2.1 High-level scanner architecture diagram 8
2.2 Example Manta graph from Python scanner 8
2.3 ECS phases high-level diagram 12
2.4 SQL scanner phases high-level diagram 14

3.1 Difference between data warehouse, lake, and lakehouse [30] . . . 16
3.2 The object model structure in Unity Catalog 18
3.3 Databricks interaction with external sources 25
3.4 Cluster settings in Databricks 33

4.1 The example of table level lineage in Unity Catalog 35
4.2 The example of column level lineage in Unity Catalog 35
4.3 Example graph from prototype 45
4.4 The DBeaver displaying Unity Catalog and Hive metastore contents 49

5.1 Standard Manta scanner architecture applied on Databricks use-case 62
5.2 Unity Catalog extraction workflow 64
5.3 Hive metastore extraction workflow 64
5.4 Repository folder structure 65
5.5 Shared context properties example in Databricks 66
5.6 Language context example 67
5.7 High-level communication diagram 69
5.8 Multiple possible values example 71
5.9 Example of variable values context 72
5.10 example of multiple contexts due to branching 73
5.11 Larger number of contexts example 75
5.12 An example of possible contexts 75
5.13 An example of all needed contexts 76
5.14 An example of reduced contexts 77
5.15 Shared context workflow 79
5.16 Notebook analysis workflow 82
5.17 Notebook calling another notebook example 83
5.18 Notebook calling another notebook workflow 84
5.19 Pin nodes placeholders example 86
5.20 Pin nodes connected example 86
5.21 Spark context node example 87
5.22 Analyzers workflow 89
5.23 Worklist over notebooks workflow 93
5.24 Error when referencing future definition 96
5.25 Data entities node hierarchy example 100
5.26 Data entities node hierarchy example 100

6.1 Hierarchy of variable values classes in shared context 103

7.1 Databricks notebook used for evaluation 113
7.2 Result graph for the evaluation example 115

A. Attachments

A.1 User Documentation

In order to run the Databricks scanner extraction and analysis, there are several requirements for the environment that have to be fulfilled:

- You need to have Java 11 installed on your computer.
- You need to have the Manta Flow platform installed on your computer. Note that this can be a major obstacle since only customers or employees of Manta have access to this program.

A.1.1 Building the project

Our code consists of the Hive metastore extraction-related classes, and all of the Dataflow Generator classes related to the analysis of notebooks using external scanners. These source codes are only a part of the whole Databricks scanner in Manta. However, all of the code that is a part of this thesis is already a part of the Manta Flow in a preview release mode. Hence, no building of the project is necessary, since the Databricks scanner is already a part of the Manta Flow installation package. During the deployment of the Databricks scanner into the Manta Flow, all modules of the Databricks scanner are built so that all module dependencies would be satisfied.

A.1.2 Running the Databricks scanner in Manta Flow

If the user has the Manta Flow platform installed on their computer, they need to do the following steps.

1. Create a connection for Databricks. In this step, the credentials to the Databricks instance have to be specified alongside the Personal Access token used for authentication.
2. Create a workflow that executes the Extraction Scenario and DataFlow Scenario. The Extraction Scenario is responsible for extracting the data to the computer and the DataFlow Scenario is used to analyze the extracted data.
3. After the execution of the scenarios is finished the results can be viewed in Manta Flow Viewer.

A.2 Contents of the Attachment

The files distributed alongside this work contain all the source codes of the Databricks scanner, images used in this work, and the Latex source codes used to create this work. Let us now describe the folder structure.

All of the images used in this thesis can be found in the *img* folder. This folder further divides the images into subfolders based on the chapter where the image has been used. There are the following subfolders:

- *analysis* - contains images used in Chapter 4, Analysis
- *background* - contains images used in Chapter 2, Data lineage analysis using Manta Flow
- *databricks-overview* - contains images used in Chapter 3, Databricks
- *design* - contains images used in Chapter 5, Design
- *evaluation* - contains images used in Chapter 7, Evaluation
- *impl* - contains images used in Chapter 6, Implementation

The Latex source codes can be found in the *tex-source* folder.

Last but not least, the source codes of the Databricks scanner can be found in the *source-code* folder. The folder further divides into sub-folders:

- *Databricks* - contains the source codes for Databricks scanner implementation
  - *Connector* - contains the source codes for the Extractor and Resolver model parts of the scanner.
  - *Dataflow* - contains the source codes for the Generator part of the scanner.
- *ECS* - contains the classes that had to be added to the Embedded Code Service for Python
- *Model* - contains classes that were added to a common Manta Flow platform model
- *Python* - contains the classes that had to be added to the Python scanner