FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Assignment of bachelor's thesis

Title:	Generic database metadata extractor
Student:	Victor Petukhov
Supervisor:	Ing. Jan Trávníček, Ph.D.
Study program:	Informatics
Branch / specialization:	Software Engineering
Department:	Department of Software Engineering
Validity:	until the end of summer semester 2023/2024

Instructions

MANTA supports the extraction of metadata information about database objects (schemas, tables, views, columns, etc.) from the most used databases systems,
Manta implements this for each database with a specific extractor, which uses the implementation details of each concrete database system,
Metadata from less common databases could be extracted using a common JDBC interface without any need to implement a specific extractor,
The goal of the thesis is to analyze the JDBC interface, propose a configurable generic extractor able to extract most of the metadata available, and implement it,
Test the generic JDBC extractor against already supported database systems for the correctness and on not yet supported database systems to evaluate the extractor's applicability.

Electronically approved by Ing. Michal Valenta, Ph.D. on 2 January 2023 in Prague.

Bachelor's thesis

GENERIC DATABASE METADATA EXTRACTOR

Victor Petukhov

Faculty of Information Technology
Software Engineering department
Supervisor: doc. Ing. Jan Trávníček, Ph.D.
May 10, 2023

Contents

List of Figures

List of Tables

List of code listings

I would like to express my deepest gratitude to my supervisor, Ing. Jan Trávníček, Ph.D., whose guidance was vitally important for my work.
I would also like to thank Ing. Ondřej Mazanec and Mgr. Jiří Toušek for their support and assistance. Their knowledge and insights have been crucial in helping me navigate the complex landscape of my research, and I am truly grateful for their contributions.
Lastly, I would like to express my sincere appreciation to my family, especially my wife, whose love, support, and understanding have been invaluable.

Declaration

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"), in accordance with Article 2373 of the Act No. 89/2012 Coll., the Civil Code, I hereby grant a nonexclusive and irrevocable authorization (license) to use this software, to any and all persons that wish to use the software. Such persons are entitled to use the software in any way without any limitations (including use for-profit purposes). This license is not limited in terms of time, location and quantity, is granted free of charge, and also covers the right to alter or modify the software, combine it with another work, and/or include the software in a collective work.

In Prague on May 10, 2023 .....

Abstract

This bachelor's thesis aims to demonstrate the feasibility of implementing a generic metadata extractor for DBMSs using JDBC API. The study implemented the JDBC metadata extractor, which can be used for all relational database management systems that provide the JDBC interface.

The motivation behind implementing a generic metadata extractor was to address the absence of a suitable generic extractor for all databases in Manta software solutions for data lineage analysis. The generic metadata extractor which uses JDBC API for gathering the metadata is implemented. The extractor is capable of retrieving various details such as the name, catalog, and schema of procedures, functions, tables, and views. Additionally, it provides information on the parameters and return types in these routines. The extractor can gather information about columns and their data types of tables, and views. The extractor was tested, and its ability to retrieve details about the database structure was demonstrated. The extracted entries are stored in in-memory or H2 databases using dictionary that was implemented using Manta dictionary abstractions.

The findings of the study demonstrated that implementing a generic metadata extractor for RDBMS using JDBC API is feasible. The implemented extractor can be used for database management systems supported by the JDBC interface. It can provide information about the database structure required for data analysis.

Further development and optimization of the extractor could lead to even more efficient and powerful database analysis.

Keywords generic database metadata extractor, JDBC API, Manta Scanner

Abstrakt

Tato bakalářská práce si klade za cíl ukázat proveditelnost implementace generického extraktoru metadat pro DBMS pomocí JDBC API. V rámci studie byl implementován JDBC extraktor metadat, který může být použit pro všechny relační databázové systémy, které poskytují JDBC rozhraní.

Motivací za implementací generického extraktoru metadat bylo vyřešit absenci vhodného generického extraktoru pro všechny databáze v softwarových řešeních Manta pro analýzu datových toků. Byl implementován generický extraktor metadat, který využívá JDBC API pro získání metadat. Extraktor je schopen získávat různé informace, jako je jméno, katalog a schéma procedur, funkcí, tabulek a pohledů. Navíc poskytuje informace o parametrech a návratových typech těchto rutin. Extraktor může získávat informace o sloupcích a jejich datových typech tabulek a pohledů. Extraktor byl testován a jeho schopnost získávat informace o struktuře databáze

byla demonstrována. Extrahované položky jsou ukládány do paměti nebo H2 databází pomocí slovníku, který byl implementován pomocí abstrakcí slovníku Manta.

Výsledky studie ukázaly, že je proveditelné implementovat generický extraktor metadat pro RDBMS pomocí JDBC API. Implementovaný extraktor může být použit pro databázové systémy podporované JDBC rozhraním. Může poskytnout informace o struktuře databáze, které jsou potřebné pro analýzu dat.

Další vývoj a optimalizace extraktoru by mohl vést k ještě efektivnější a výkonnější analýze databáze.

Klíčová slova generický extraktor metadat z databáze, JDBC API, Manta Scanner

Abbreviations

JDBC	Java Database Connectivity
DBMS	Database Management System
RDBMS	Relational Database Management System
API	Application Programming Interface
MSSQL	Microsoft SQL Server
SSL	Secure Sockets Layer
DDL	Data definition language
URL	Uniform Resource Locators
POM	Project Object Model
DTO	Data Transfer Object
CSV	Comma-Separated Values

Introduction

Manta Tools is a company that specializes in providing software solutions for data lineage analysis. Data lineage is the process of tracking data from its origin to its final destination, which is essential for data governance, compliance, and auditing purposes. Manta Tools' software products enable businesses to track and visualize data lineage across various data sources, platforms, and technologies, providing them with visibility into their data and enhancing their decision-making capabilities. In order to achieve this, Manta Tools utilizes an extractor system, which extracts data from various sources, and a resolver system, which resolves complex data relationships to provide a complete and accurate view of data lineage. The extractor and resolver are critical components of Manta Tools' software solutions, allowing businesses to gain insights into their data and make informed decisions about data governance, compliance, and auditing.

One of the critical components of the software solution for data lineage analysis is an extractor. An extractor is a component that is responsible for pulling data from data stores and feeding it into the data lineage system for analysis. While Manta Tools does provide many database-specific extractors, they do not have a generic extractor that is suitable for all databases. To address this, the possibility of implementing the generic metadata extractor was analyzed. What's more, the implementation of the metadata extractor was done using JDBC API. While the JDBC extractor may not be as powerful as a database-specific extractor, it can be used for all database management systems that provide the JDBC interface, including those that do not have a custom extractor available.

Chapter 1
Requirements analysis

1.1 Problem statement

Some databases are unsupported by Manta. The extractor was not created for them. Manta needs the generic extractor for all databases to make it possible to extract information about entities that exist in the given database regardless of the existence of the database's connector.[1].

1.2 Functional requirements

This section describes functional requirements need to be fulfilled to develop the generic extractor.

1. Establish a connection to the database by providing a URL, a username, and a password.
2. Extract metadata, which includes information about the types and names of entities present in the database.
3. Be able to apply include and exclude filters to extract specific catalogs and schemas of the database.
4. Store the extracted data in a suitable format.
5. Make extraction possible for both Manta-supported and unsupported DBMSs.

1.3 Non-functional requirements

This section describes non-functional requirements need to be considered during the development of the generic extractor.

1. Store extracted metadata in either an H2 database or an in-memory dictionary.
2. Integrate extractor in the manta platform.

[1] unit that includes extractor and resolver for the specific DBMS in Manta platform

Chapter 2
Analysis

This chapter includes an analysis of the possibility of implementing the generic JDBC metadata extractor. The use of applicable technologies is analyzed. The important knowledge for the future realization of the project is presented in the chapter.

2.1 Entities to be extracted

The particular entity types of the database system are extracted.

Catalog is a collection of objects. It represents the highest level of organization in a database and can contain schemas.

Schema is a logical container that holds objects such as tables, views, procedures, synonyms, and functions. It represents a way to organize and group related database objects together.

Table is a collection of related data organized in rows and columns.

View is a virtual table that is based on the result of a database query. It displays information in a customized format, without altering the original data.

Procedure is a set of instructions which takes some input and performs certain tasks. It represents a way to encapsulate and reuse code within a database. Usually, procedures do not return a value. However, the procedure may have a return value if specific instructions are used.

Function is similar to the Procedures. Nevertheless, the function return value should be specified.

Synonym is an alias for a database object such as a table, view, function or procedure. It represents a way to simplify and shorten the name of an object.[1]

2.2 Understanding the capabilities of the JDBC

The JDBC API is a powerful tool for Java developers to interact with relational databases. It provides a standard way for Java applications to connect to and perform operations on a wide range of databases.

[1] synonym extraction is omitted. Explained in Section 2.6.12

2.2.1 Limitations of the JDBC API

The JDBC API is exclusively designed for communication between Java programs and relational databases. The JDBC API does not include any built-in features for facilitating interactions with NoSQL databases. Therefore, the JDBC API is entirely devoted to working with relational systems.[1]

2.3 DataSource interface and BasicDataSource class

The DataSource interface is part of the Java Database Connectivity API. It is used in Java applications to provide a standard way of managing database connections and connection pooling, as well as to abstract away the details of connecting to different types of databases.[2]

The connection properties required for establishing a connection to the data source are set using the DataSource interface. This includes information such as the host name, port number, database name, user credentials, and any other required parameters for the specific data source. The DataSource class provides a standard way to manage these connection properties, and it can be easily integrated with various database technologies, such as MySQL, PostgreSQL, Oracle, and Microsoft SQL Server.

The BasicDataSource class is an implementation of the DataSource interface that provides connection pooling. Connection pooling is a technique that involves creating a pool of database connections that can be reused by multiple threads or processes in an application. By reusing connections from the pool, the application can avoid the overhead of establishing new connections each time data access is required. When using the `getConnection` method provided by BasicDataSource, the pool will return a connection if one is available. If there are no available connections, the pool will create a new connection. However, the limitation on new connections can be set.

The BasicDataSource class will be used in the testing and analysis phases to establish a connection with a database.

2.3.1 Connecting to the database

In order to extract the metadata from every RDBMS, a generic way of connecting to every relational database should be found.

For the non-secured connection, a connection string, driver name, username, and password should be provided. The username and password are credentials for entering the database. The driver name depends on the RDBMS. A driver name is provided in order to load a specific driver. A JDBC driver allows Java applications to interact with a database. The connection string specifies the hostname and port number of the SQL server instance that you want to connect to. An example of the MSSQL connection string is `jdbc:sqlserver://mssql.int.getmanta.com:1433`. The host name and port number are important for establishing a connection with the right database.

BasicDataSource class provides methods including `setDriverClassName(String drivName)`, `setUrl(String url)`, `setUsername(String userName)`, `setPassword(String password)` for setting the connection parameters.

For the secured connection, certain parameters should be configured in addition to the non-secured connection parameters. The parameters of a MSSQL connection string are used as an example to show how a secured connection must look. To enable SSL encryption for the connection, the `"encrypt=true"` parameter should be configured.[3] Integrated security must be disabled with the `"integratedSecurity=false"` parameter. This means that the SQL Server will not use the Windows credentials of the user running the Java application. To ensure that the certificate presented by the SQL Server is trusted, the `"trustServerCertificate=false"`

parameter should be used to instruct the driver not to blindly trust the certificate. Instead, the trust relationship should be established in the usual way. The `"hostNameInCertificate"` parameter specifies the expected hostname in the SSL certificate presented by the SQL Server. Finally, the `"trustStore"` parameter specifies the location of the trust store that contains the trusted SSL certificates, and the `"trustStorePassword"` parameter specifies the password for the trust store. These parameters ensure that the certificate presented by the server is trusted and verified.

In some cases, the server may require that you present a valid client certificate in order to establish a connection. This is known as client-side authentication, and it provides an additional layer of security to ensure that only authorized clients can access the server.[4]

Establishing the secured connection can be done either through the connection string or through the parameters of the BasicDataSource class.

It is possible that some databases may not support establishing a secured connection through the BasicDataSource class or may have specific requirements or limitations on how to establish a secured connection. Nonetheless, the connection string should be a generic solution for establishing an encrypted connection.

In order to prove the assumption, the list of dialects supporting encrypted connections in Manta was tested. The list includes Mssql, Oracle, PostgreSQL, Teradata, Hive, and Db2. HIVE, DatabricksSQL, and MSSQL were tested practically. The proof of the assumption about establishing the secured connection through the connection string of PostgreSQL is found on the "Initializing the Driver" official page of PostgreSQL.[5] Similarly, the proof for DB2 DBMS was found online in the "HOW TO: Connect to SSL enabled DB2 through the JDBC V2 connector in IICS" article.[6] For Oracle DBMS, the proof was found in the Oracle official blog.[7] Finally, it was proven for Teradata under the "Making a Database Connection" section on the page "Teradata JDBC Driver".[8]

Based on these findings, it can be concluded that the connection string approach is a viable solution for establishing secure connections across the multiple DBMS used by Manta. It is likely that the connection string approach can be used to establish secure connections across all the databases that support encrypted connections. Therefore, it can be concluded that the connection string approach is a generic and reliable solution for establishing secure connections across various DBMSs with encrypted connections.

2.4 DatabaseMetaData interface

The DatabaseMetaData class in Java is an important part of the JDBC API that provides a way for Java programs to retrieve metadata information about a database. This metadata information includes details about the database structure, such as tables, columns, indexes, stored procedures, and more. Metadata refers to information about the structure and properties of the database rather than the data stored in the database.

The DatabaseMetaData class provides a set of methods that can be used to retrieve this metadata information. The important methods for the JDBC metadata extractor are listed in Figure 2.1.[9]

The `getCatalogs` method retrieves the information of the catalogs available in the given database.

The `getSchemas` method retrieves information about the schemas.

The `getProcedures` method retrieves a description of the stored system and user procedures.

The `getProcedureColumns` method retrieves a description of the given stored procedure parameter and result columns.

The `getFunctions` method retrieves a description of the stored system and user functions.

The `getFunctionColumns` method retrieves a description of the given function parameters and return type.

The `getTables` method retrieves a description of tables, views, synonyms, and more.

The `getColumns` method retrieves a description of columns of entities extracted using the `getTables` method.

Every observed method in Java returns a ResultSet java object.

Figure 2.1 Included methods of the DatabaseMetaData class

```
getCatalogs();

getSchemas(String catalog, String schemaPattern);

getProcedures(String catalog, String schemaPattern, String procedureNamePattern);

getProcedureColumns(String catalog, String schemaPattern, String procedureNamePattern,
String columnNamePattern);

getFunctions(String catalog, String schemaPattern, String functionNamePattern);

getFunctionColumns(String catalog, String schemaPattern, String functionNamePattern,
String columnNamePattern);

getTables(String catalog, String schemaPattern, String tableNamePattern, String[] types);

getColumns(String catalog, String schemaPattern, String tableNamePattern,
String columnNamePattern);
```

2.4.1 Excluded methods

There are several methods that were excluded from being used but may be important for a future enhancement of the JDBC Scanner.

The method `getPrimaryKeys` retrieves a description of the primary key columns of the given table.

The method `getExportedKeys` retrieves a description of the foreign key columns that reference the given table's primary key columns. The foreign keys exported by a table.

The method `getImportedKeys` retrieves a description of the primary key columns that are referenced by the given table's foreign key columns. The primary keys imported by a table.

Overall, these methods provide essential information about the table relationships in a database, and the JDBC metadata extractor can use them to build an understanding of the database schema.

Figure 2.2 Excluded methods of the DatabaseMetaData class

```
getPrimaryKeys(String catalog, String schema, String table);

getExportedKeys(String catalog, String schema, String table);

getImportedKeys(String catalog, String schema, String table);
```

2.4.2 Filtering database metadata

All methods, except for `getCatalogs()`, can be filtered by specifying the catalog name as the first parameter of the method.

In certain methods of the DatabaseMetaData class, you may need to provide pattern arguments as strings. These patterns use special characters to determine which metadata to retrieve.

The "%" character in a pattern string represents any number of characters, including none, while the "_" character represents a single character. These pattern arguments are used to filter the metadata entries, and only the entries that match the search pattern will be returned.

If you set any argument in the method signature to `null`, it will be excluded from the search criteria, meaning that it won't be used to filter the metadata entries. This can be useful for creating more customized search queries based on specific criteria.[9]

2.5 Result Sets of the methods of the DatabaseMetaData class

In Java, a ResultSet is an object that represents a set of results obtained from executing a SQL statement. It is part of the Java Database Connectivity API and is used to retrieve data from a database.[10]

A ResultSet object maintains a cursor that points to the current row in the result set. By default, the cursor is positioned before the first row, and the `next()` method is used to move the cursor to the next row. The ResultSet is typically used in a loop to process each row of the result set until there are no more rows. For example, the outcome of invoking the `getTables` method would consist of a collection of information about the retrieved tables. Each individual row in the collection would represent a distinct table.

The ResultSet object provides methods for accessing the data in the current row, such as `getInt`, `getString`, `getDouble`, and so on. The parameters of the methods can be either column index, represented by an integer, or column label, represented by a string. In the context of the JDBC Scanner[2], the `getInt`, `getString` methods are useful because the data retrieved after the invocation of the methods from Figure 2.1 can be either a string type or an integer type.

2.5.1 Closing ResultSet object

It is important to close the ResultSet object after you have finished using it to release database resources and prevent memory leaks.

The connection should not be forgotten and closed. What's more, the connection should not be closed if the ResultSet object is still used. It may throw a SQLException exception. The best practice is to close all the objects related to the connection before the connection is closed.

2.6 Differences in Result Sets in distinct databases

During the execution of the Java program and the retrieval of data from a SQL database, it was observed that certain columns in the ResultSet were throwing a SQLException exception with the message "Invalid column name". It was determined that this was due to the specific implementation of the ResultSet attributes in different databases, which may result in some columns being unavailable in certain databases.

To address this issue, the ResultSetMetaData class was utilized to retrieve information about the columns in the ResultSet that are accessible in different databases. This class provides metadata about the ResultSet object, including the number of columns, the column names,

[2]another name for the JDBC metadata extractor

Code listing 2.1 Usage of the ResultSetMetaData class for analysis of the Result Sets

```
ResultSetMetaData metaData = resultSet.getMetaData();
                int columnCount = metaData.getColumnCount();
                for (int i = 1; i <= columnCount; i++) {
                    String columnName = metaData.getColumnName(i);
                    String columnType = metaData.getColumnTypeName(i);
                    System.out.println("Column␣" + i + ":␣" + columnName
                    + "␣(" + columnType + ")" + "Value:␣"
                    + resultSet.getString(columnName));
                }
```

data types, and other properties. The usage the ResultSetMetaData class can be observed in Listing 2.1

The investigation was carried out on three databases, namely Oracle, Microsoft SQL Server, and PostgreSQL. These databases are provided by Manta's development environment.

The ResultSetMetaData class was used to retrieve the column names and their corresponding data types for each of these databases. The results indicated that there are differences in the columns that were accessible across the three databases.

2.6.1 Comparison of the retrieved catalogs metadata

In this analysis, we compare the output of the `getCatalogs()`.

In Oracle, the database structure consists of a two-segment hierarchy. There are no catalogs in the Oracle database structure. Therefore, the ResultSet object after the invocation of the `getCatalogs()` method on the Oracle database is empty.

PostgreSQL has only one database, while MSSQL has many databases.

The retrieved data conforms to the structures of the databases. The generic column for all three databases is "TABLE_CAT". No more columns were found.

2.6.2 Comparison of retrieved schemas metadata

In this analysis, we compare the output of the `getSchemas` with null parameters. The columns and their associations with the databases are listed in Table 2.1.

In Oracle, when executing the `getSchemas(null,null)` method, all existing schemas in the database are extracted with a catalog name equal to `null`. The output includes some default schemas of the Oracle database, such as "SYS".

In PostgreSQL, executing the method with `null` arguments also shows all existing schemas in the database without a catalog name. The catalog name is `null`. The output includes the schema "information_schema," which provides a standardized view of metadata for all objects within a database.

In MSSQL, executing the method `getSchemas("ENT\_ODS",null)` is expected to return one schema in the catalog "ENT_ODS". However, the actual output contains many default system schemas, such as "db_accessadmin" and "db_ddladmin". When executing the method with `null` arguments, the output only shows system schemas and schemas from the master catalog, which also contains system schemas. Nevertheless, there are more catalogs than one in the database.

The generic columns for all three databases are "TABLE_CATALOG" and "TABLE_SCHEM". System schemas are extracted with every extraction of schemas in all three DBMSs. In Oracle and PostgreSQL, when executing `getSchemas(null,null)` all existing schemas in the database are displayed without catalog. In Oracle, such behavior is expected. Nevertheless, the information about user-defined schemas in the PostgreSQL database should have been retrieved with the

catalog names. Moreover, the number of extracted user-defined MSSQL schemas when executing `getSchemas(null,null)` is smaller than expected.

2.6.3 Comparison of retrieved procedures metadata

In this analysis, we compare the output of the `getProcedures` with null parameters. The columns of the output are illustrated in Table 2.1.

When executing the `getProcedures(null,null,null)` method in Oracle, the output contains nine columns. The "SPECIFIC_NAME" column is always `null`, and three columns are undefined, always returning `null`. The column label[3] of these columns is "NULL".

In PostgreSQL, the output contains the same columns as Oracle, but the three undefined column are named as "?column?", always returning `null`.

In MSSQL, executing the same method returns eight columns. The columns from six to eight always return -1 for all procedures.

The five columns are common to all three databases. MSSQL doesn't have any information about a specific name of the procedure. The additional columns and their information in the output vary between Oracle, PostgreSQL, and MSSQL. Oracle and PostgreSQL have three undefined columns, while MSSQL includes three columns with different labels and fixed information, presumably intended for future use.

Table 2.1 Procedures metadata in different databases

	Column	Database
1	PROCEDURE_CAT	generic
2	PROCEDURE_SCHEM	generic
3	PROCEDURE_NAME	generic
4	REMARKS	generic
5	PROCEDURE_TYPE	generic
6	NUM_INPUT_PARAMS	MSSQL
7	NUM_OUTPUT_PARAMS	MSSQL
8	NUM_RESULT_SETS	MSSQL
9	SPECIFIC_NAME	Oracle and PostgreSQL

2.6.4 Comparison of retrieved procedure parameters and return type metadata

The current examination involves a comparison of the output results of the `getProcedureColumns` with `null` arguments. The columns of the Result Sets are illustrated in Table 2.2.

In Oracle, the columns "COLUMN_DEF", "SCALE", "SQL_DATA_TYPE", "SQL_DATETIME_SUB" and "DEFAULT_VALUE" are always `null` in the tested data. Four columns are database-specific columns. Twenty-three columns are extracted.

In PostgreSQL, "PRECISION", "LENGTH", "SCALE", "RADIX" "REMARKS", "COLUMN_DEF", "SQL_DATA_TYPE", "SQL_DATETIME_SUB", "CHAR_OCTET_LENGTH" are always `null` in the tested data. Twenty columns are extracted in total.

In MSSQL, nine columns are specific columns of the MSSQL database. The columns from twenty to twenty-seven in Table 2.2, "COLUMN_DEF", and "SQL_DATETIME_SUB" are always `null` in the tested data. Twenty-eight columns are extracted.

[3]column label is a name of the column of the Result Set row

There are nineteen common columns across all three databases. According to the DatabaseMetaData interface documentation "SQL_DATETIME_SUB" and "SQL_DATA_TYPE" are unused. However, there is information in the "SQL_DATA_TYPE" column in MSSQL. The "SPECIFIC_NAME" is not retrieved as the column of the Result Set in MSSQL comparing to other databases. "COLUMN_DEF" is `null` in all of the databases according to the tested data.

Table 2.2 Procedure parameters and return type metadata in different databases

	Column label	Database
1	PROCEDURE_CAT	generic
2	PROCEDURE_SCHEM	generic
3	PROCEDURE_NAME	generic
4	COLUMN_NAME	generic
5	COLUMN_TYPE	generic
6	TYPE_NAME	generic
7	PRECISION	generic
8	LENGTH	generic
9	SCALE	generic
10	RADIX	generic
11	SQL_DATA_TYPE	generic
12	SQL_DATETIME_SUB	generic
13	CHAR_OCTET_LENGTH	generic
14	ORDINAL_POSITION	generic
15	IS_NULLABLE	generic
16	NULLABLE	generic
17	REMARKS	generic
18	COLUMN_DEF	generic
19	DATA_TYPE	generic
20	SS_TYPE_CATALOG_NAME	MSSQL
21	SS_TYPE_SCHEMA_NAME	MSSQL
22	SS_UDT_CATALOG_NAME	MSSQL
23	SS_UDT_SCHEMA_NAME	MSSQL
24	SS_UDT_ASSEMBLY_TYPE_NAME	MSSQL
25	SS_XML_SCHEMACOLLECTION_CATALOG_NAME	MSSQL
26	SS_XML_SCHEMACOLLECTION_SCHEMA_NAME	MSSQL
27	SS_XML_SCHEMACOLLECTION_NAME	MSSQL
28	SS_DATA_TYPE	MSSQL
29	SPECIFIC_NAME	Oracle and PostgreSQL
30	SEQUENCE	Oracle
31	OVERLOAD	Oracle
32	DEFAULT_VALUE	Oracle

2.6.5 Comparison of retrieved functions metadata

In this analysis, we compare the output of the `getFunctions` method with `null` arguments. The columns of the retrieved Result Set are listed in Table 2.3.

The method with `null` parameters in Oracle returns six columns and the "SPECIFIC_NAME" column is always `null`.

When this method is executed on the PostgreSQL database, `getFunctions(null,null,null)` returns 6 columns. Unlike Oracle, the "SPECIFIC_NAME" column is not `null` in PostgreSQL,

and it provides a unique name for each function.

In the case of MSSQL, eight columns are extracted, and three database-specific columns always have a value of -1.

In conclusion, there are 5 common columns in all three databases.

Table 2.3 Functions metadata in different databases

	Column	Database
1	FUNCTION_CAT	generic
2	FUNCTION_SCHEM	generic
3	FUNCTION_NAME	generic
4	REMARKS	generic
5	FUNCTION_TYPE	generic
6	NUM_INPUT_PARAMS	MSSQL
7	NUM_OUTPUT_PARAMS	MSSQL
8	NUM_RESULT_SETS	MSSQL
9	SPECIFIC_NAME	Oracle and PostgreSQL

2.6.6 Comparison of retrieved function parameters and return type metadata

The present analysis compares the output results of the `getFunctionColumns` with `null` arguments. The columns of the Result Sets are illustrated in Table 2.4.

In Oracle, the method returns twenty-three columns with some of them being specific to the Oracle database. Some columns are always `null` in the tested data, including “SCALE”, “COLUMN_DEF”, “SQL_DATA_TYPE”, “SQL_DATETIME_SUB”, “SPECIFIC_NAME”, and “DEFAULT_VALUE”.

In PostgreSQL, the method returns seventeen columns, and some of the columns that are always `null` in the tested data include “PRECISION”, “LENGTH”, “SCALE”, and “RADIX”.

In Microsoft SQL Server, the method returns twenty-eight columns. In the tested data, the columns “COLUMN_DEF”, “SQL_DATETIME_SUB”, and columns from twenty to twenty-seven in Table 2.4 are always `null`.

There are sixteen columns that are common for all three databases. Most of the database-specific columns of MSSQL has `null` values.

2.6.7 Comparison of retrieved tables metadata

The present analysis compares the output results of the `getTables` with the first two parameters equal to `null` and the latter parameter equal to `String[]{"TABLE"}`. Therefore, all entities of type table are extracted. The output columns are listed in Table 2.5.

For Oracle, the five columns are extracted.

In contrast, when executing `getTables(null, null, null, String[]{"TABLE"})`, ten columns are extracted from PostgreSQL. All `null` values are found in all the database-specific columns of the PostgreSQL database.

In MSSQL, only five columns are extracted, similar to Oracle.

In conclusion, there are five common columns for all three databases. “TABLE_TYPE” column is always equal to `TABLE`, because the types parameter in the method is assigned to `TABLE` and the output was filtered accordingly.

Table 2.4 Function parameters and return type metadata in different databases

	Column label	Database
1	FUNCTION_CAT	generic
2	FUNCTION_SCHEM	generic
3	FUNCTION_NAME	generic
4	COLUMN_NAME	generic
5	COLUMN_TYPE	generic
6	TYPE_NAME	generic
7	PRECISION	generic
8	LENGTH	generic
9	SCALE	generic
10	RADIX	generic
11	SQL_DATA_TYPE	generic
12	SQL_DATETIME_SUB	Oracle and MSSQL
13	CHAR_OCTET_LENGTH	Oracle and MSSQL
14	ORDINAL_POSITION	generic
15	IS_NULLABLE	generic
16	NULLABLE	generic
17	REMARKS	generic
18	COLUMN_DEF	Oracle and MSSQL
19	DATA_TYPE	generic
20	SS_TYPE_CATALOG_NAME	MSSQL
21	SS_TYPE_SCHEMA_NAME	MSSQL
22	SS_UDT_CATALOG_NAME	MSSQL
23	SS_UDT_SCHEMA_NAME	MSSQL
24	SS_UDT_ASSEMBLY_TYPE_NAME	MSSQL
25	SS_XML_SCHEMACOLLECTION_CATALOG_NAME	MSSQL
26	SS_XML_SCHEMACOLLECTION_SCHEMA_NAME	MSSQL
27	SS_XML_SCHEMACOLLECTION_NAME	MSSQL
28	SS_DATA_TYPE	MSSQL
29	SPECIFIC_NAME	Oracle and PostgreSQL
30	SEQUENCE	Oracle
31	OVERLOAD	Oracle
32	DEFAULT_VALUE	Oracle

Table 2.5 Tables metadata in different databases

	Column	Database
1	TABLE_CAT	generic
2	TABLE_SCHEM	generic
3	TABLE_NAME	generic
4	REMARKS	generic
5	TABLE_TYPE	generic
6	TYPE_CAT	PostgreSQL
7	TYPE_SCHEM	PostgreSQL
8	SELF_REFERENCING_COL_NAME	PostgreSQL
9	REF_GENERATION	PostgreSQL
10	TYPE_NAME	PostgreSQL

2.6.8 Comparison of retrieved synonyms metadata

The current examination involves a comparison of the output results of the `getTables(null, null, null, String[]{"SYNONYM"})` with the first two parameters equal to `null` and the latter parameter equal to `String[]{"SYNONYM"}`. Hence, all entities of the type synonym are extracted.

The same columns were extracted for Oracle and MSSQL as in Section 2.6.7.

PostgreSQL does not offer functionality that is equivalent to SQL Server synonyms.[11] Therefore, the observed Result Set of the retrieved data from the PostgreSQL database was empty.

2.6.9 Comparison of retrieved views metadata

The current examination involves a comparison of the output results of the `getTables(null, null, null, String[]{"VIEW"})` with the first two parameters equal to `null` and the latter parameter equal to `String[]{"VIEW"}`. Therefore, entities of the type view are extracted.

The same columns were extracted for all three databases as in Section 2.6.7.

The database-specific columns of the Posgtresql were extracted and columns “TYPE_CAT”, “TYPE_SCHEM”, “TYPE_NAME”, “SELF_REFERENCING_COL_NAME”, and “REF_GENERATION” have `null` values likewise.

2.6.10 Comparison of retrieved table columns metadata

The current examination involves a comparison of the output results of the `getColumns` with `null` arguments. The columns of the output results are presented in Table 2.6.

In Oracle, the columns “SCOPE_CATALOG”, “SCOPE_SCHEMA”, “SCOPE_TABLE”, “SQL_DATA_TYPE”, “SQL_DATETIME_SUB” and “SOURCE_DATA_TYPE” are always `null` in the tested data. Twenty-four columns are extracted in total.

Similar to Oracle, twenty-four columns are extracted from the PostgreSQL database, and the columns with `null` values are identical.

In MSSQL, all the database-specific columns, “COLUMN_DEF”, “SCOPE_SCHEMA”, “SCOPE_CATALOG”, “SQL_DATETIME_SUB”, and “SCOPE_TABLE” are always `null` in the tested data. Thirty-two columns are extracted.

In summary, there are twenty-four columns generic for all three databases. However, “SCOPE_CATALOG”, “SCOPE_SCHEMA”, and “SCOPE_TABLE” are always `null` in all the tested data sources. “COLUMN_DEF” is usually `null` but can have different values in some cases. In the tested data, “SOURCE_DATA_TYPE” is always present in the MSSQL database. “SQL_DATA_TYPE” and “SQL_DATETIME_SUB” are always `0` or `null` in the tested data for Oracle and PostgreSQL. According to the DatabaseMetaData interface documentation, the columns “SQL_DATA_TYPE” and “SQL_DATETIME_SUB’ are unused. However, different values can be retrieved from the MSSQL database for these columns.

2.6.11 “SPECIFIC_NAME” and “COLUMN_DEF” columns absence

The “SPECIFIC_NAME” is a column of the methods: `getProcedures`, `getFunctions`, `getProcedureColumns`, and `getFunctionColumns`. It is a default column according to the DatabaseMetaData interface documentation. Nevertheless, there is no such column in MSSQL.

The “COLUMN_DEF” is a column of the methods: `getProcedureColumns`, `getColumns`, and `getFunctionColumns`. It is a default column according to the DatabaseMetaData interface

Table 2.6 Table columns metadata in different databases

	Column label	Database
1	TABLE_CAT	generic
2	TABLE_SCHEM	generic
3	TABLE_NAME	generic
4	COLUMN_NAME	generic
5	COLUMN_SIZE	generic
6	TYPE_NAME	generic
7	BUFFER_LENGTH	generic
8	DECIMAL_DIGITS	generic
9	NUM_PREC_RADIX	generic
10	DATA_TYPE	generic
11	SQL_DATA_TYPE	generic
12	SQL_DATETIME_SUB	generic
13	CHAR_OCTET_LENGTH	generic
14	ORDINAL_POSITION	generic
15	IS_NULLABLE	generic
16	NULLABLE	generic
17	REMARKS	generic
18	COLUMN_DEF	generic
19	SCOPE_CATALOG	generic
20	SCOPE_SCHEMA	generic
21	SCOPE_TABLE	generic
22	SOURCE_DATA_TYPE	generic
23	IS_AUTOINCREMENT	generic
24	IS_GENERATEDCOLUMN	generic
25	SS_IS_SPARSE	MSSQL
26	SS_IS_COLUMN_SET	MSSQL
27	SS_UDT_CATALOG_NAME	MSSQL
28	SS_UDT_SCHEMA_NAME	MSSQL
29	SS_UDT_ASSEMBLY_TYPE_NAME	MSSQL
30	SS_XML_SCHEMACOLLECTION_CATALOG_NAME	MSSQL
31	SS_XML_SCHEMACOLLECTION_SCHEMA_NAME	MSSQL
32	SS_XML_SCHEMACOLLECTION_NAME	MSSQL

documentation. However, it was not extracted when the `getFunctionColumns` was executed on the PostgreSQL database.

2.6.12 Not enough information about the database synonyms

It is not possible to extract enough information about synonyms with the methods `getTables` and `getColumns`. Firstly, the `getTables` method extracts information about a catalog name and schema name, where a synonym is located, and the name of the synonym. The absence of information about the entity to which the synonym refers, such as information about the entity's name, kind, and location, makes it impossible to determine its reference entity and connect the entity with the synonym. Secondly, the `getColumns` method gives no information on synonym columns. It would not be important if the first point was covered correctly. Otherwise, it could give us a synonym that is constructed as a table. It would be better than having no information

about the synonyms. According to the second point, creating a synonym as a table is not possible without having any information about columns.

There is no rudimentary information about synonyms, such as a reference entity type, name, location, or columns of the synonyms. The decision is not to extract synonyms in any way.

2.6.13 Summary and additional information

During the analysis, many database-specific columns were retrieved. Especially, a large number of columns specific to the MSSQL database were obtained during the analysis.

Many columns of the Resut Sets were always `null`. It is likely that most of these columns were created but remained unused, given the large volume of data that was tested.

It is worth mentioning that there are different data types of the retrieved data. Nevertheless, it is converted to the suitable Java data type format. For example, for the column "COLUMN_DEF" the data type is `LONG` in oracle and `varchar` in PostgreSQL. It is converted to the String data type in Java.

Testing the extracted data for each database is crucial to ensuring that any default columns that may be missing are identified. What's more, additional columns vary from one database to another.

2.7 Relevant columns of the Result Sets

After the comparison of the columns of the Result Sets, the generic columns were found for all three databases. With assistance from the documentation of DatabaseMetaData interface and output of the analyses, the most relevant for the JDBC metadata extractor columns of the Result Sets were chosen.

2.7.1 Useful columns of the retrieved catalog information

The only column is retrieved using `getCatalogs` method. "TABLE_CAT" contains a catalog name in a Java String format.

2.7.2 Useful columns of the retrieved schema information

The two columns are observed after the `getSchemas` invocation. "TABLE_CATALOG" carries information about the catalog name where the schema is located, and "TABLE_SCHEM" has a schema name. Both of the columns have a string format.

2.7.3 Useful columns of the retrieved procedure information

There are four relevant columns found in the output Result Set of the method `getProcedures`.

The first column, "PROCEDURE_CAT" specifies the catalog name where the procedure can be found. The second column, "PROCEDURE_SCHEM", specifies the schema name where the procedure is located. The third column, "PROCEDURE_NAME", is a string that identifies the name of the procedure. All three columns have a string format. Finally, the "PROCEDURE_TYPE" column provides information on whether the procedure returns a value or not. The column is described in Section 2.7.10. The "PROCEDURE_TYPE" column is of the integer data type.

2.7.4 Useful columns of the retrieved procedure column information

The Result Set of the `getProcedureColumns` contains several columns relevant for the JDBC metadata extractor that provide information about each column in the procedure. The first three columns, "PROCEDURE_CAT", "PROCEDURE_SCHEM", and "PROCEDURE_NAME", provide information about the location of the procedure and the name of the procedure. It can be useful for finding the exact procedure to which the column belongs. The "COLUMN_NAME" attribute specifies the name of the column. These four columns are of type string. The "COLUMN_TYPE" attribute provides information about the role of the column. The column is of the integer type and described in Section 2.7.11. Finally, the "TYPE_NAME" attribute provides the name of the data type of the column. The "DATA_TYPE" attribute specifies the SQL type of the column using an integer value from the java.sql.Types class. The columns are explained in Section 2.7.14 and in Section 2.7.13, accordingly.

2.7.5 Useful columns of the retrieved function information

There are three relevant columns found in the output Result Set of the method `getFunctions`.

The first column, "FUNCTION_CAT" specifies the catalog name where the function can be found. The second column, "FUNCTION_SCHEM", specifies the schema name where the function is located. The third column, "FUNCTION_NAME", is a string that identifies the name of the function. All three columns have a string format.

2.7.6 Useful columns of the retrieved function column information

The Result Set of the `getFunctionColumns` contains several columns relevant for the JDBC metadata extractor that provide information about each column of the function.

The columns "FUNCTION_CAT", "FUNCTION_SCHEM", and "FUNCTION_NAME" represent a location and a name of the function to which the column belongs. All three columns are of type string. The "COLUMN_NAME", "COLUMN_TYPE", "TYPE_NAME" and "DATA_TYPE" columns have the same meaning as in the Result Set output of the `getProcedureColumns`.

2.7.7 Useful columns of the retrieved table information

The output Result Set of the method `getTables` contains four columns with relevant information. The first column is called "TABLE_CAT" and indicates the catalog name where the table is located. The second column is called "TABLE_SCHEM" and specifies the schema name where the table can be found. The third column is named "TABLE_NAME" and contains a string that identifies the name of the table. All three columns have a string format. The last column, "TABLE_TYPE", is a string value that contains the name of the type of the table. The column is described in Section2.7.9.

2.7.8 Useful columns of the retrieved table column information

The `getColumns` method is used to retrieve several important columns for the JDBC Scanner from the Result Set. The columns "TABLE_CAT", "TABLE_SCHEM", and "TABLE_NAME" represent a location and a name of the table of the current column, while "COLUMN_NAME" indicates the name of the column. These four columns have a string type. The "DATA_TYPE"

column, which contains information about the SQL type from java.sql.Types, is of the integer type. “TYPE_NAME” indicates the name of the data type. The “TYPE_NAME” column is of the string type. Finally, “NULLABLE” contains information about whether the column allows `null` values. The “NULLABLE” is of type short, and it is described in Section2.7.12

2.7.9 “TABLE_TYPE” column

The “TABLE_TYPE” column is found in the Result Sets of the method `getTables`. “TABLE_TYPE” contains information about a table type. It can be useful to distribute entities and store them separated by a table type. Some types that might be useful include “SYNONYM”, “VIEW”, and “TABLE”.

2.7.10 “PROCEDURE_TYPE” column

The “PROCEDURE_TYPE” column is found in the Result Sets of the method `getProcedures`. “PROCEDURE_TYPE” contains information about whether a procedure has a return type. This column can take one of the three possible values: unknown, does not return a value, or returns a value. These values are assigned to the numbers 0, 1, and 2, respectively.

The storage of a procedure with information about its signature can benefit from knowing whether a saved procedure has a return type or not.

2.7.11 “COLUMN_TYPE” column

The Result Sets of the methods `getProcedureColumns` and `getFunctionColumns` have the “COLUMN_TYPE” column. It indicates a role of the column. It can take one of six possible values: unknown, input, input/output, output, return value, or result column in `ResultSet`. These values are assigned to the numbers 0 to 5, respectively.

The role of the column is important because it determines how the column is used within the routine[4]. What’s more, it determines how it is expected to behave. For example, an input column is used to pass data into a routine, while an output column is used to return data from a routine.

2.7.12 “NULLABLE” column

The “NULLABLE” column is found in the Result Sets of the method `getColumns`. “NULLABLE” contains information about whether the column allows `null` values. There are three possible values: 0, 1, or 2, representing the options that the column cannot contain null values, the column can contain null values, or nullability is unknown, respectively.

The nullability of the column affects the behavior of the database when inserting, updating, and querying data. For example, when inserting data into a table, if a column is not nullable, then a value must be provided for that column. Otherwise, an error will occur.

The nullability is an important piece of additional information for the columns.

2.7.13 “DATA_TYPE” column

The “DATA_TYPE” column is found in the Result Sets of the methods `getProcedureColumns`, `getFunctionColumns`, `getColumns`. It gives information about the SQL type from java.sql.Types. Specifically, a column returns a constant value from the Types class. This is very helpful for the JDBC Scanner implementation as it generalizes all the data types from different RDBMSs.

[4]routine is another name for a function or a procedure

For example, Oracle data type `VARCHAR2`, PostgreSQL data type `name`, and MSSQL data type `nvarchar` are generalized to the Java SQL type `Types.NVARCHAR` constant from the java.sql.Types class.

Having such information plays a crucial role in representing data types in the generic extractor.

2.7.14 "TYPE_NAME" column

The "TYPE_NAME" column is found in the Result Sets of the methods `getProcedureColumns`, `getFunctionColumns`, `getColumns`. "TYPE_NAME" indicates a name of the data type. The name is database-specific. For example, a PostgreSQL type name may return `int4` or `varchar` for the primitive data types. Moreover, the column can return more information when it comes to complex data types, such as structure. For example, in PostgreSQL, the `ddltest_t1` can be returned as a type name of the data type `Types.STRUCT`. The column can be useful for showing more information about complex data types.

What's more, SQL type `Types.OTHER` does not give us much information. The `Types.OTHER` indicates that the type is database-specific and cannot be generalized as others[12]. The type name may be helpful in this situation too. It may give a name of the data type that is used in the database. For example, for the data type `Types.OTHER` type name "record" was given in PostgreSQL as the return type of the procedure using `getProcedureColumns`. With the knowledge of a database-specific type name, it can help to understand how the data is stored and how it can be accessed or manipulated.

2.7.15 Ancestor name columns

The information about ancestor names such as "PROCEDURE_CAT", "PROCEDURE_SCHEM", and "PROCEDURE_NAME" in the Result Set of the `getProcedureColumns` method or "TABLE_CATALOG" in the Result Set of the `getSchemas` method can be very important for validating that the entities that are extracted are the descendant of the exact ancestors that are provided.

2.7.16 Column and Schema names accept null values

In the Result Set of the methods `getColumns`, `getProcedureColumns`, `getFunctionColumns`, `getProcedures`, `getFunctions`, and `getTables`, columns that express schema or catalog name such as "PROCEDURE_SCHEM" or "TABLE_CAT" can be `null` values. The column that indicates the catalog name of a schema in the Result Set of the `getSchemas` can be equal to `null`. The `null` values of the catalogs and schemas are observed during the extraction of the system entities.

A catalog name being `null` value is also observed in the extraction of the entities inside the catalog in the PostgreSQL database. It can be marked as invalid metadata extraction, and it is illustrated in Section 2.8.5.

2.7.17 Excluded columns

There are several columns that were excluded from being used in the JDBC Scanner. However, they may be important for the future or require some explanation as to why they were removed from being used.

The column "ORDINAL_POSITION" is extracted using the `getColumns`, `getProcedureColumns`, `getFunctionColumns` methods. It is used to indicate a position of the parameter in a function or a procedure or an index of the column in a table. For instance, if a

stored procedure has three input parameters, the "ORDINAL_POSITION" column would have the values 1, 2, and 3, respectively, indicating the position of each parameter in the procedure. This information is particularly useful when creating DDL statements, as it helps ensure that correct parameters are assigned to correct positions in a procedure. However, recreation of DDL scripts is not part of the JDBC Scanner. Nevertheless, it may be important for the future if generic recreation of DDL scripts is considered possible and important.

A value of "SCOPE_CATALOG" pertains to a catalog of the table that serves as scope of the reference attribute. Similarly, a value of "SCOPE_SCHEMA" pertains to a schema of the table that serves as scope of the reference attribute. Lastly, a value of "SCOPE_TABLE" pertains to a name of the table that serves as scope of the reference attribute. A value of the columns is `null` if the data type `Types.REF` is not defined. The column "DATA_TYPE" with the value `Types.REF` was not presented in any extracted metadata of the columns using methods `getColumns`, `getProcedureColumns`, `getFunctionColumns`. Due to the absence of a specific data type, it has been decided that the columns "SCOPE_CATALOG", "SCOPE_SCHEMA", and "SCOPE_TABLE", which are associated with the REF SQL data type, will not be extracted.

The column "FUNCTION_TYPE" that is extracted using `getFunctions` provides information on whether a function returns a table. It can have one of three values: unknown, does not return a table, or returns a table, which are assigned numerical values of 0, 1, and 2, respectively. It is recommended to consider the use of the column in future implementations of the JDBC Scanner, as it can help understand the return type of the function more thoroughly.

The "COLUMN_DEF" attribute is extracted as a Result Set column of the methods `getColumns` and `getProcedureColumns`. "COLUMN_DEF" indicates the default value of the column. `null` value of the column means that the default value is either not specified or `null` itself.

The "PRECISION" column indicates the maximum number of digits or characters that can be stored in a specific data type column. The "LENGTH" column provides the length of a column's value, in bytes or characters. The "SCALE" column indicates the number of digits that can be stored to the right of the decimal point in a numeric column. These three columns are extracted using `getProcedureColumns` or `getFunctionColumns`.

The maximum amount of data that a column can store is defined by the "COLUMN_SIZE" column. The column is extracted using `getColumns` method.

It is recommended to consider the use of the columns "COLUMN_DEF", "PRECISION", "LENGTH", "SCALE", and "COLUMN_SIZE" in future implementations of the JDBC Scanner as it gives extra information about the column.

2.7.18 Summary

Enough information was found to implement the generic JDBC scanner. The database structure can be recreated, and many additional pieces of information can be retrieved, especially for the columns of the routines, tables, and views.

2.8 Examples of specific JDBC API extractor behavior

In this section, the JDBC API atypical and unforeseen actions are examined and explained, along with the examination of obscure test scenarios.

2.8.1 Extraction of the catalogs and schemas in the Teradata database

The choice was made to test the Teradata database as well because of its unique hierarchical structure. There are no schemas in Teradata's two-segment hierarchy. A DBMS with the same

hierarchy has not been tested yet. Due to this, it was decided to test the Teradata for information extraction about catalogs and schemas using the DatabaseMetaData class and its methods `getCatalogs` and `getSchemas` with `null` arguments.

The method `getCatalogs` has no catalogs in the database. In the meantime, the `getSchemas` method has several schemas. It was discovered that the JDBC extractor interprets databases as schemas even though Teradata has databases but no schemas.

2.8.2 Crash-prone substring in the Teradata database

During the extraction of the Teradata database's entities, the crash-prone view name is found in the tested data. There is a name of the view "ugly" \:/*" which is in the database for the purpose of testing edge-cases. The method failed to read columns of the view using `getColumns`. The SQLException was thrown by the method. The error message was about invalid escape clause "\:". The table name was specified as a parameter.

Efforts, such as escaping special characters, were made to prevent special characters from causing issues with the name, but none of them were effective in preventing an exception from occurring.

2.8.3 Extraction of the system schemas

The `getSchemas(catalog_name,null)` extracts all the schemas of the given catalog and system schemas that do not have a catalog. The behavior is unexpected given that the catalog name ought to narrow down the search. However, it leaves the system schemas in the output.

It is considered a general behavior observed in all three tested dialects.

2.8.4 Incorrect Extraction using methods with null arguments

It is important to observe that, as per the official documentation of the DatabaseMetaData interface, "`null` means the catalog name should not be used to narrow down the search."[9] However, when executing the `getSchemas(null, null)` method, it does not show all of the catalogs, and consequently not all the schemas. Such behavior can be observed in MSSQL.

`get***`[5] methods with all `null` arguments ought to extract all the entities of the type *** in the given database. However, it is not always the case, as `getSchemas(null,null)` does not always extract all schema's catalogs, as it is observed in Section 2.6.2. Only entities that are located outside of any catalog or inside those catalogs which schemas are extracted using `getSchemas(null,null)` method are extracted with all the null arguments.

Therefore, utilizing `get***` methods with `null` arguments does not allow for the retrieval of all of the data that is stored in the database. It highlights the importance of considering the output of the methods using `null` arguments.

2.8.5 Incorrectly extracted information about the PostgreSQL database's schemas

In PostgreSQL, the output of the `getCatalogs()` method is equal to the Result Set with one row. The row has one column "TABLE_CAT" which is equal to the name of the catalog. The `getSchemas` method with the catalog name parameter retrieves only the schemas with catalog equal to `null`. It extracts the schemas from the correct catalog (regardless of system schemas). Nevertheless, it shows the catalog name as `null` value.

[5] *** stands for an entity type

In conclusion, some user-defined schemas can be extracted without the catalog name, even when the schema has one.

2.8.6 Incorrect Extraction of the Oracle database hierarchy

In Oracle, the catalog name column is equal to `null` because there are no catalogs. However, there is an exception.

Oracle has packages. Package is a schema object that groups logically related PL/SQL types, variables, constants, subprograms, cursors, and exceptions. The Oracle database hierarchy consists of a schemas, which in turn contain packages and some information contained within those packages. An example of the hierarchy is in Figure 2.3. The database administration tool was used to obtain the information. As it can be observed, "TEST_SYNONYMS_PROC" is the entity inside the "TEST_SYNONYMS" package. Every package is located inside the "DWH" schema. Despite that, the JDBC extractor shows a different hierarchy when procedures are trying to be extracted using `getProcedures` method. The Result Set has columns "PROCEDURE_CAT", "PROCEDURE_SCHEM", and "PROCEDURE_NAME" where the catalog name is "TEST_SYNONYMS", schema name is "DWH", and the procedure name is "TEST_SYNONYMS_PROC". Hence, the hierarchical structure is thoroughly disordered. In the meantime, there was no schema with the catalog "TEST_SYNONYMS" retrieved when the `getSchemas(null,null)` method was executed.

In conclusion, there may be some discrepancies in a hierarchy of database objects when they are extracted through the DatabaseMetaData class. These discrepancies could potentially cause confusion or errors.

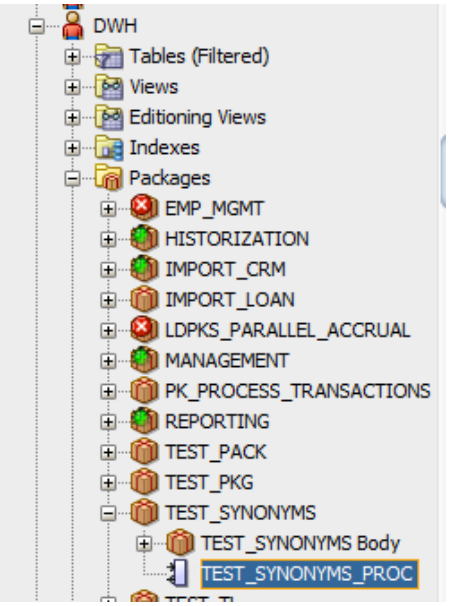

Figure 2.3 The hierarchy of the package in the Oracle database

2.8.7 Retrieval of columns containing referential data types

In order to understand how the DatabaseMetaData methods extracts columns with a reference type, a test case was produced.

A table “t1” has two columns. The first column is of type `int` and has the name “a”. The second column is of type `int` and has the name “b”. A second table with the name “t2” has a column “x” with the type `t1`. The type is a reference to the table “t1”. The question is how the data is extracted. The “t2” table will either have two columns of type integer or one column of type “t1”.

The case is tested with the PostgreSQL database. The output of the `getColumns(catalog_of_table_t2,schema_of_table_t2,t2_name,null)` is a Result Set with one column. The “COLUMN_NAME” is “x”. The “DATA_TYPE” of the column “x” is equal to the `Types.Struct`. The name of the table “t1” is assigned to the column “x” as the “TYPE_NAME” value.

In conclusion, the data is extracted by fetching the column with the reference type rather than extracting the reference type columns.

2.9 Filtering database and schema entities to be extracted

The class DatabaseSchemaFilter provides a two-level filtering mechanism for databases and schemas based on regular expressions. The class is created by Manta and it is used in the Connectors. It allows specifying include and exclude filters in the format of “database/schema, database/schema, etc.”, where each comma-separated value represents a database and schema combination to filter. The format can be different if the DBMS has a two-segment hierarchy. In that situation, the filter is “database, database, etc.” or “schema, schema, etc.” depending on what entity type is excluded from the hierarchy.

Chapter 3
Design

The goal for the JDBC Scanner is to be consistent with other Connectors of Manta. The same frameworks, libraries, tools and structures ought to be used. To ensure consistency and compatibility with other connectors, modules structure and their interconnections are going to be examined.

3.0.1 Design constraint

The extractor must maintain consistency in the classes and methods used, as well as preserve the same structure as in other extractors. This requirement ensures that the new extractor is built in accordance with existing standards, enabling easier integration with other components of the system. By maintaining consistency in the classes and methods used, the new extractor can leverage existing code and promote code reuse, resulting in a more maintainable and efficient system. Preserving the same structure also promotes a more modular design, making it easier to add and modify functionality in the future.

What's more, the same frameworks, libraries and tools ought to be used. Using them across different extractors in the system also helps to ensure consistency and compatibility between components. This approach can help to reduce the likelihood of errors or conflicts that may arise when using different technologies, and can facilitate collaboration among developers who are familiar with the same technologies.

Overall, this constraint ensures that the new extractor is developed in a manner that is consistent with existing standards and promotes ease of use, maintainability, and interoperability with other system components.

3.1 Tecnnologies

There are several technologies used in Manta that are considered compulsory for the JDBC Scanner.

3.1.1 Maven

Maven is a widely used build automation tool for Java projects that helps to manage project dependencies and streamline the build process. One of the key features of Maven is its support for modular development, which allows developers to break a large project into smaller, more manageable modules that can be developed and tested independently. Maven uses a project

object model to define the structure and dependencies of a project, and each module in the project can have its own POM file.

In a Maven-based project, modules are typically interconnected through dependencies defined in the POM files. Each module can declare its own dependencies on other modules or external libraries, and Maven will automatically download and include the necessary dependencies in the build process. This allows developers to focus on developing individual modules without worrying about the dependencies of other modules in the project.[13]

3.1.2 Spring framework

Spring is a popular open-source framework for building Java applications.

One of the key features of Spring is its support for the creation and management of beans, which are objects that form the core of a Spring application. Beans are typically defined as Java classes that are managed by the Spring container, which is responsible for creating, configuring, and managing instances of those classes.

Spring provides several mechanisms for creating and configuring beans, including XML configuration files, annotations, and Java-based configuration. The XML configuration files and annotations are decided to be used for the implementation. With XML configuration files, developers can define the structure and dependencies of beans using a set of XML tags and attributes. Annotations, on the other hand, allow developers to define the configuration of beans directly in the source code.

Another key benefit of using Spring for the JDBC metadata extractor is its support for unit testing and integration testing. Spring provides a variety of tools and utilities that can be used to write effective tests for Spring-based applications. One of the ways that Spring helps during testing is by providing a lightweight container for managing the application's components. It allows to define and wire up the application's dependencies in a way that is easy to test. During testing, the container inject any necessary dependencies, making it easier to write isolated and focused tests. Spring provides a variety of testing annotations and utilities that make it easier to write effective tests.[14]

3.1.3 Junit 5

JUnit 5 is a modern and highly capable testing framework that provides with a wide range of features.

Using a testing framework is important for several reasons. It allows developers to write automated tests that can quickly and reliably validate the correctness of their code. This can help catch bugs early in the development process, reducing the time and effort required to fix them. Testing frameworks also make it easier to maintain and refactor code over time, as tests can help ensure that changes do not introduce new issues or regressions. Finally, testing frameworks can help ensure that code is robust and reliable, increasing the overall quality and value of the software being developed.

3.2 Modules interconnection

Manta introduces its maven modules dependency system. The system is generic for most of the Connectors. The idea behind the structure of the maven modules dependencies is to have all the main modules be dependent only on the one module "manta-connector-xxx-model". By using this approach, the implementation time can be reduced, and the likelihood of errors, especially in large projects, can be reduced.

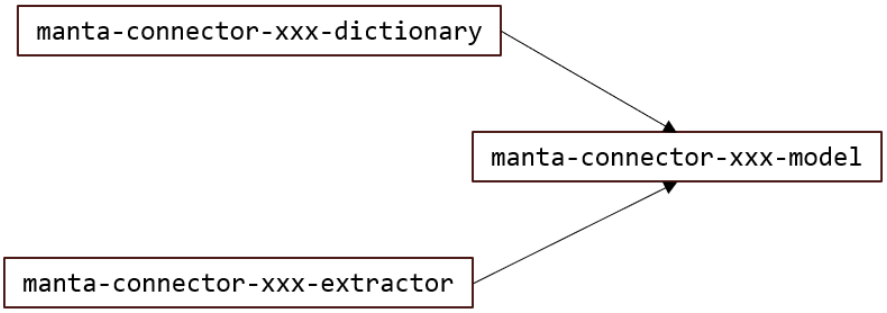

Figure 3.1 Structure of the JDBC Scanner

The "manta-connector-xxx-model"[1] contains classes that are used in several other modules and logically do not belong to any of the modules they are used in. Moreover, the module contains interfaces of classes that logically belong to another module but are used as dependencies in classes of other modules where the class of the interface is not implemented.

3.3 Modules structure

There are three modules considered useful for the JDBC Scanner implementation. The first module is "manta-connector-xxx-model". This module contains auxiliary interfaces and classes for working with extracted data and providing the dependency management system introduced in Section 3.2. The second module is "manta-connector-xxx-dictionary-extractor". This module provides an extraction scenario. The third module is "manta-connector-xxx-dictionary". This module provides access to the storage of an extracted metadata database. It provides functionality to store the data correctly, both hierarchically and syntactically, in the storage.

Using the 3 modules structure helps to simplify the implementation of the JDBC Scanner by providing clear and distinct responsibilities for each module. This can make it easier to manage dependencies and modify or add new features to the scanner in the future. The modules provide a clear separation of concerns for handling the extraction of data.

The three modules are named "manta-connector-jdbc-model", "manta-connector-jdbc-dictionary-extractor", and "manta-connector-jdbc-dictionary", accordingly.

3.3.1 POM files hierarchy

The modules are placed in the JDBC folder. Every module has its own POM file for establishing a configuration. The POM file of the module is connected to the parent POM "manta-platform-parent-pom" of all the Manta's modules. The parent POM file contains all the versions of dependencies and all the dependencies that are valid for usage during development.

The centralized dependency management helps to ensure consistency and avoid conflicts between dependencies and their versions used in different modules.

[1] xxx replaced by the name of the Connector

3.4 Common Manta modules

There are several Manta modules that provide generic logic for all of the Connectors. These modules have the name structure "manta-connector-common-xxx"[2].

The JDBC Scanner implements the dictionary module. The "manta-connector-common-dictionary" is relevant for it. It provides functionality for saving dictionary objects in the in-memory dictionary or in the H2 dictionary.[15].

The extractor module of the JDBC Scanner can use the help of the "manta-connector-common-dictionary-extractor".

Finally, the model module can use the "manta-connector-common-dictionary-model" as a dependency.

Those dependencies are helping to maintain reusability and standardization.

[2] xxx replaces by the name of the connector logic part

Chapter 4
Implementation

The implementation chapter provides an overview of the classes that have been developed and integrated into the project. This chapter details the specific decisions that were made during the implementation process and how they were carried out. Furthermore, this section covers the automated tests that were conducted to ensure the functionality and reliability of the implemented system. By the end of this chapter, readers will gain an understanding of the technical aspects of the project.

4.1 Classes of the "manta-connector-jdbc-model"

4.1.1 JdbcResolverEntitiesFactory interface

The interface called JdbcResolverEntitiesFactory is used to create resolved objects that represent a database.

The interface contains several methods for creating different types of database objects, such as catalogs and schemas, and for getting information about the database structure and built-in data types. The `createCatalog` method creates a new catalog object with the specified name in the dictionary and source type. The `createSchema` method creates a new schema object with the specified name in the given catalog object and source type. Other `create***` entity methods are created already. They are located in the interface ResolverEntitiesFactory. The JdbcResolverEntitiesFactory interface implements the interface ResolverEntitiesFactory. The `getDbStructure` method returns a description of the database structure. Finally, the `getBuiltinDataType` method returns a built-in dictionary data type that corresponds to the provided constant from the java.sql.Types class.

4.1.2 JdbcDictionaryFactory interface

The interface is used to create a memory dictionary for a database. The interface contains a single method called `createMemoryDictionary` that returns an instance of JdbcResolverEntitiesFactory.

4.2 Classes of the "manta-connector-jdbc-dictionary"

4.2.1 JdbcDialect class

The JdbcDialect class is located in the dialect package. The purpose of this class is to define which entity types and relationships are legal in a database system. The class extends the AbstractDialect class.

The JdbcDialect class contains a private field called dbStructure, which is an instance of the DbStructureImpl class. This field is used to store the structure of the database system and is used by the various methods in the class to define the legal entity types and relationships.

The JdbcDialect creates database structure during the constructor invocation using a DbStructureImpl class and its methods that define the various entity types and relationships that are legal in a database system. These include the `allows` method, which specifies which entity types are allowed to be children of a parent entity, and the `policy` method, which specifies the policy for duplicate children, and the `mismatchPolicy` method, which specifies policy for inserting an unauthorized type of child

The class also contains a `detectBuiltinSymbols` method, which is used to detect the presence of built-in entities in the given dictionary. The method is called during the dictionary object creation. The `initBuiltinSymbols` method is used to add built-in entities to the given dictionary if they are not detected. The built-in entities become the standard, persist-able part of the data dictionary.

4.2.2 JdbcDictionaryFactory class

The class is located in the "jdbc" package. It provides static methods for creating and accessing a JDBC data dictionary. The class contains four static methods. `openDictionary` for opening an existing dictionary, `reinitDictionary` for creating a new initialized dictionary and deleting any existing data, and two overloaded versions of both methods that take either a JdbcDictionaryDataSource or a DataSource instance. The class creates a dictionary in the H2 database.

4.2.3 JdbcDictionarySource class

The class is located in the "jdbc" package. This is a Java class that implements the DictionarySource interface. It contains methods for obtaining an input stream, an output stream, and a JDBC data dictionary object.

The DictionarySource interface is used later by the ConnectorTask, as it is documented in the code.

4.2.4 MemoryDictionaryFactory class and JdbcMemoryDictionaryFactoryImpl class

The class MemoryDictionaryFactory is located in the "memory" package. It provides static methods for creating and accessing a JDBC data dictionary. The class contains two static methods. `openDictionary` for opening an existing dictionary, `reinitDictionary` for creating a new initialized dictionary and deleting any existing data.

The class JdbcMemoryDictionaryFactoryImpl creates a dictionary in the in-memory database with the Jdbc dialect.

4.2.5 JdbcDataDictionary class

JdbcDataDictionary is a class that extends AbstractDataDictionary and also implements JdbcResolverEntitiesFactory. Its purpose is to store and persist data in the data dictionary, which needs to be maintained between the extraction and analysis phases. In AbstractDataDictionary,

the `create***` methods are used to store dictionary objects. JdbcDataDictionary's constructor has four parameters: the EntityDAO class, the Dialect class, a String representing the global namespace[1] (dictionary id), and a boolean value that indicates whether the dictionary should be initialized. Initialization involves deleting all data in the data store, creating built-in symbols if any exist, and persisting the current state.

To create a specific database dictionary, there are mandatory constructor and methods that need to be implemented.[15] The constructor must call `setCaseSensitive` to set the case-sensitivity of the dictionary, and `createDatabase` and `createSchema` methods are necessary for saving a catalog and schema, respectively. JdbcDataDictionary's `getItemType` method decides the generic dictionary item type for an entity. It may return `null` for non-persistent entities. Finally, the `getDbStructure` method should be implemented.

4.3 Classes of the "manta-connector-jdbc-dictionary-extractor"

4.3.1 Entity package

There is an "entity" package that includes DTO classes. They are used as temporary storage for the extracted data.

The Catalog class is a DTO that represents a catalog, which contains a `name` field and a list of `schemaNames`. The list of `schemaNames` represents schema names in the catalog with the name that is represented with the field `name`.

The Entity class is an abstract class that serves as a base for other entity DTOs. It has three fields: `catalog`, `schema`, and `name`.

The Routine class is a subclass of the Entity class and represents a database routine, such as a stored procedure or function. In addition to the fields inherited from Entity, the class has three additional fields: `parameters`, `returnColumns`, and `unknownColumns`. These fields are lists of routine columns that express the signature of a routine and the existence of undetermined columns.

The Function class is a subclass of the Routine class and represents a database function. It does not add any additional fields or methods. The class is used to enhance the code's readability.

The Procedure class is a subclass of the Routine class and represents a database procedure. It has a `hasReturnValue` field, which is a Java Boolean value indicating whether or not the procedure returns a value. The `hasReturnValue` field can be set to `null` if it is unknown, `false` if it does not return a value, or `true` if it does return a value.

The Table class is a DTO that represents a table or a view. It extends the Entity class. The Table class has two additional fields: `type` and `columns`. The `type` field is of an enumeration type[2] TableType. The `columns` field is a list of Column objects representing the columns in the table or view.

The TableType class represents an enumeration type. It contains two constants, `TABLE` and `VIEW`, which represent the types that can be extracted from a database. Each constant has a corresponding string value that can be used to identify the type. The constructor of the TableType enumeration takes a string parameter that is used to initialize the type field of the constant. The TableType enumeration is used to specify the type of a Table entity.

The Column class is a DTO for a database column. It has four private fields: `name` representing the name of the column, `dataType` representing the data type of the column stored as a constant from java.sql.Types, `canBeNull` representing if the column allows `null` values, and `typeName` representing the database-specific type name of the column. There are three possibilities for the field `canBeNull`: the column can contain `null` values, the column cannot contain

[1] global namespace is a top level name of the database hierarchy. Catalogs are saved under the global namespace
[2] represents a set of predefined constants

`null` values, or the nullability is unknown. These three possibilities are assigned values of `true`, `false`, and `null`, respectively.

The ColumnType class represents an enumeration of column roles in a database. It contains a set of constants representing different roles that a column can have, such as `"IN"`, `"OUT"`, `"RETURN\_VALUE"`, `"RESULT\_SET"`, and `"UNKNOWN"`. It has a constructor that takes a String parameter to set the name of the role. The class also includes a map that associates integer values with the corresponding column roles.

The RoutineColumn class is a DTO used to represent a column in a database procedure or function. It extends the Column class and adds an additional field `role` of type ColumnRole that represents the role of the column in the routine.

The ColumnInformativeAttributes class is just a placeholder for the constants that are used to store the information about the nullability of the column into the dictionary using the DictionaryWriterImpl class.

4.3.2 MetaDao interface

This is an interface called MetaDao, which stands for "metadata access object". It defines methods for extracting metadata about various objects from a database and storing them as DTOs.

The interface includes methods for extracting catalogs, tables, views, procedures, and functions from a database, as well as methods to filter extracted data.

The `getTables`, `getProcedures`, and `getFunctions` methods take two parameters: the name of the catalog and the name of the schema. These methods return a list of DTOs for the corresponding type of object that exists within the specified catalog and schema. The method `getAllCatalogs` has no parameters. It creates a list of catalog DTOs. The `getFilteredCatalogsWithSchemas` method returns a filtered list of Catalog DTOs, where the filter criteria are specified by the `includeFilter` and `excludeFilter` properties.

4.3.3 MetaDaoImpl class

The class MetaDaoImpl is an implementation of the MetaDao interface with various methods to extract metadata information about a database using the DatabaseMetaData class.

The MetaDaoImpl class uses a DataSource instance to establish a connection to the database. Moreover, include and exclude filters are fields of the class.

4.3.4 DictionaryWriter interface

The DictionaryWriter interface is responsible for writing entities into a dictionary within a specific global namespace. The interface defines several methods for writing different types of objects into the dictionary, such as catalogs, schemas, procedures, tables, views, and functions.

The interface has several methods for writing objects into the dictionary. These methods have the name structure `write*`. The interface has a method called `writeCatalog` that takes a String argument as a catalog name and writes the catalog into the dictionary. The interface has a method called `writeSchema` that takes a schema name and an IResDataType argument, which expresses a dictionary object of the saved catalog. The method writes the schema into the dictionary. Both `writeSchema` and `writeCatalog` return a dictionary object representing the saved entity. The `writeProcedure` and `writeFunction` methods both take in a dictionary schema object and a procedure or function DTO, respectively, and save the routine. The `writeProcedureWithoutParams` and `writeFunctionWithoutParams` methods are similar to the `writeProcedure` and `writeFunction` methods, but they generate a routine signature without any parameters or return values. The `writeTableOrView` method receives a dictionary object of

a schema and a table DTO and writes a table or view into the dictionary based on the type of the table. Finally, the `persistChanges` method persists the changes made to the dictionary.

4.3.5 DictionaryWriterImpl class

DictionaryWriterImpl class implements all the methods of the DictionaryWriter interface.

The purpose of the DictionaryWriterImpl class is to write database objects and their data types into a dictionary. The DictionaryWriterImpl class uses a JdbcResolverEntitiesFactory object to represent the dictionary.

4.3.6 JdbcExtractor interface

This is an interface called JdbcExtractor. It defines methods for extracting entities and saving them to a dictionary. The main method is `extract()` which extracts all the entities and saves them to the dictionary. It is the main method of the whole project.

When all the important parameters and fields are initialized, the `extract()` method can be invoked to produce the desired scenario.

4.3.7 JdbcExtractorImpl class

The JdbcExtractorImpl class is a Java class implementing the JDBC extractor interface.

The class has a MetaDao and a DictionaryWriter instance, which are used to extract metadata and save objects to the dictionary, respectively. It also has a dictionaryId instance variable, which is a global namespace for all the extracted entities.

4.3.8 JdbcExtractorReader class

The JdbcExtractorReader class is located in the scenario package and implements the InputReader interface for JdbcResolverEntitiesFactory. The purpose of this class is to extract the Jdbc dictionary using the JdbcExtractor and save the entities from the data source.

The class has four public methods. The `canRead()` method returns `true` as it is always able to read input. The `read()` uses the JdbcExtractor instance to extract the dictionary and return it. The `close()` has an empty implementation as there are no resources to release. The `getInputName()` returns the name of the global database in the extracted dictionary.

These four public methods override methods of the InputReader interface, which is used in other parts of Manta’s application.

4.3.9 Auxiliary classes

The ConvertUtils class provides static methods to convert certain metadata information, extracted from a database, into values that are used to fill the fields of the corresponding DTOs. Specifically, `columnNullableIntToBool` method converts a `nullable` value of a column from an integer, retrieved from metadata, into a Boolean, used in the column DTO. The method returns true if the column can be null, false if it cannot be null, and null if it is unknown. `procedureTypeIntToBool` converts the type of a procedure from an integer into a Boolean, indicating whether the procedure has a return value. The method returns true if the procedure has a return value, false if it does not have a return value, and null if it is unknown. The class has a private constructor, indicating that it is not meant to be instantiated.

Code listing 4.1 Part of the getAllCatalogs() method

```
while (resultSet.next()) {
    String catalogName = resultSet.getString("TABLE_CAT");
    Catalog catalog = new Catalog(catalogName);
    catalogs.add(catalog);
}
if (catalogs.isEmpty()) {
    Catalog catalog = new Catalog("");
    catalogs.add(catalog);
```

4.4 Other packages

4.4.1 Categories package

The "categories" package has a Category class inside. What's more, it has an errors sub-package, which has classes for generating error messages for the logger.

Manta uses its own logging API. The special plugin should be installed for generating error classes out of the methods introduced in the error package classes. Methods in the Category class create new instances of the error classes in order to use them for logging.

You can find the Category package implemented in the "manta-connector-jdbc-dictionary-extractor" and "manta-connector-jdbc-dictionary" modules.

4.5 Implementation decisions

4.5.1 Generic extraction of catalogs and schemas

There are different hierarchies and structures of schemas and catalogs inside the database. Usually, there is a three-segment hierarchy structure: catalog, schema, and other entities. However, there can be a two-segment hierarchy structure with no schemas, as in the Teradata DBMS, or with no catalogs,as in the Oracle DBMS. The extraction of catalogs and schemas in Oracle and Teradata can be found in Section 2.8.6 and Section 2.8.1, respectively.

In order to omit retrieving packages in Oracle and other potential redundant data in other two-segment hierarchy databases, the methods with the catalog name parameter put the value "" empty string as a catalog name. That makes the extraction possible only for schemas and entities inside it with the catalog name equal `null`.

In order to know when the database does not have either catalogs or schemas, the output of the method `getCatalog()` should be considered. If `getCatalog()` returns empty Result Set, there are no catalogs or schemas. For saving the two-segment hierarchy, it occurs to me to use the same approach as for excluding potentially redundant data. The empty string for the catalog name is used. Schemas are representing a top level hierarchy in the saved database with one catalog with the empty string name.

For PostgreSQL and MSSQL databases, catalogs are retrieved using `getCatalog()` method of the DatabaseMetaData class and all the schemas are retrieved using `getSchemas(catalog_name,null)` are assigned to the related catalog.

Taking into account everything written in the current section, the part of the public method `getAllCatalogs()` in Listing 4.1 is implemented in MetaDaoImpl class. It extracts data about the catalogs from the given connection. If there are no catalogs, it adds a catalog with an empty string in order to write the schemas in the catalog.

It is worth mentioning that schemas with the `null` catalog name are saved for every catalog due to issue in Section 2.8.5.

4.5.2 Filtering of the catalogs and schemas

There are two public static overloaded methods named `includeExclude` in the class called DatabaseSchemaFilter. They are listed in Figure 4.1. The first is a static method that determines whether a given database and schema combination should be included in the extraction or excluded from it based on provided include and exclude filters. That method is useful for a DBMS with a three-segment hierarchy. The second is another static method that does the same as the first method, but instead of the combination, it takes a schema or a database name. Regardless of the parameter named "database" in the second method in Figure 4.1, the schema name can be put as an argument and filtered accordingly. That method is useful for a DBMS with a two-segment hierarchy.

The include and exclude filters must be initialized beforehand.

Figure 4.1 Methods for filtering extraction of schema and database entities

```
includeExclude(DatabaseSchemaFilter includeFilter, DatabaseSchemaFilter excludeFilter,
String database, String schema);

includeExclude(DatabaseSchemaFilter includeFilter, DatabaseSchemaFilter excludeFilter,
String database);
```

Three-segment hierarchy DBMSs can be filtered using a catalog name and schema name. The first method shown in Figure 4.1 is applied. Only the schema name is supplied as the parameter for two-segment hierarchy databases, including Teradata, Oracle, and others. The second method shown in Figure 4.1 is applied.

What is more to take into account is that entities have the ability to throw an exception with a message indicating a lack of privileges for reading inside the specific catalog. The catalogs should be filtered before extracting other entity types in order to avoid accessing entities inside the catalog with specific rights. The two-segment hierarchy follows a similar pattern. However, in that circumstance, the schemas ought to be filtered. The filtering of catalogs or schemas is done using the second method shown in Figure 4.1.

Additionally, system schemas are extracted. By utilizing the exclude and include filters, the user is responsible for their exclusion.

Taking into account everything written in the current section, the public method in Listing 4.2 is implemented in the MetaDaoImpl class. Firstly, it uses the public method `getAllCatalogs()` for extracting all catalogs. If the database has both catalogs and schemas, it filters the catalogs, retrieves the schemas of the filtered catalogs, and then filters the schemas. If the database has either catalogs or schemas, it stores schemas under one catalog with the name "" empty string and filters them afterward.

4.5.3 Routines with the same name

If a database has two functions or procedures with the same name in the same schema and catalog, their columns are extracted without any information about which column belongs to which routine. These routines are referred to as overloaded. An illustration would be when the first function has the name "function_name" and the column name is "first_column". The second function in the same location has the name "function_name" likewise. The column name of the second function is "second_column". The "first_column" and "second_column" columns are extracted for both functions, as there is no parameter to specify in the method `getFunctionColumns` to extract only one of these functions. The `getFunctionColumns(catalog_name,schema_name,function_name,null)` method has the same arguments specified for both of the columns.

Code listing 4.2 Filtering of the catalogs and schemas

```
@Override
public List<Catalog> getFilteredCatalogsWithSchemas() {
    List<Catalog> catalogs = getAllCatalogs();
    if (hasCatalogs(catalogs)) {
        List<Catalog> filteredCatalogs =
        getFilteredCatalogs(catalogs);
        storeSchemasInCatalogs(filteredCatalogs);
        List<Catalog> doubleFilteredCatalogs =
        getFilteredSchemas(filteredCatalogs, true);
        return doubleFilteredCatalogs;
    } else {
        storeSchemasInCatalogs(catalogs);
        List<Catalog> filteredCatalogs = getFilteredSchemas(catalogs,
        false);
        return filteredCatalogs;
    }
}
```

A comparison of values of the "SPECIFIC_NAME" column from the Result Set of the `getFunctionColumns` and `getProceudureColumns` methods may be the answer to solving the problem for the PostgreSQL database. We can learn more about the unique name of the routine from "SPECIFIC_NAME" column. This solution, however, only works with PostgreSQL. "SPECIFIC_NAME" has always `null` value in the tested data for the Oracle database, and the column does not exist in the Result Set of the MSSQL database.

The general approach is to properly record routines with unique names, including their parameters and return values. However, no information regarding the parameters or return type should be stored for overloaded procedures and functions. It is unknown how many overloaded routines there are in a database. Therefore, we simply record one routine without parameters for them. The implementation of the above-mentioned problem is carried out in the JdbcExtractorImpl class and can be observed in the method in Listing 4.3. A similar method is implemented for the procedures. The method retrieves database functions and divides functions that are overloaded and those that are not overloaded into two groups using the `routinesWithUniqueNames` method. The functions that are not overloaded are passed to `writeFunction` of the DictionaryWriter class, and they are written with all the information retrieved. Otherwise, `writeFunctionWithoutParams` of the DictionaryWriter class is used. The function is written without any parameters or return type in that situation.

4.5.4 Avoiding naming patterns

There are a few patterns, like "%" and "_". It signifies that any occurrence of "%" and "_" characters in the name of a table, schema, procedure, or function will violate the meaning of such a name during the extraction of its children.

An example would be a table with the name "vo%". We wish to extract columns from the "vo%" table. However, the method `getColumns(catalog_name, schema_name, "vo%", null)` retrieves all columns from tables whose names begin with "vo". Overly many columns can be extracted.

The solution to the problem is presented in Listing 4.4. The method of the MetaDaoImpl class compares a given schema name and a given table name with a retrieved schema and table name for every column. If the given and retrieved names are equal, the column is stored. Otherwise, the column is not stored in the table's DTO.

Code listing 4.3 Method for the extraction and saving functions

```
private void extractAndWriteFunctions(IResDataType dictSchema,
String catalogName, String schemaName) {
    List<Function> functions =
    metaDao.getFunctions(catalogName, schemaName);
    List<Function> uniqueFunctions = new ArrayList<>();
    List<Function> notUniqueFunctions = new ArrayList<>();
    routinesWithUniqueNames(functions, uniqueFunctions,
    notUniqueFunctions);
    for (Function function : uniqueFunctions) {
        dictionaryWriter.writeFunction(dictSchema, function);
    }
    for (Function function : notUniqueFunctions) {
        logger.log(Categories.extractionErrors().
        routineColumnsNotStored()
                .catalog(function.getCatalog())
                .schema(function.getSchema())
                .name(function.getName()));
        dictionaryWriter
        .writeFunctionWithoutParams(dictSchema, function);
    }
}
```

In general, the name of an ancestor we have and the name of an ancestor retrieved using the Result Set are compared. The same solution can be applicable for procedures, functions, and schemas. Every time the pattern parameter exists in the method of the DatabaseMetaData class and it is desired to retrieve children of the specific database entity, it is important to compare the parent's name with the parent name that was taken from the child.

It is worth mentioning that catalog names cannot be compared. In some databases, if a catalog exists and an entity is located inside it, the catalog name, after retrieving the entity's data, can be equal to `null`. It is considered in Section 2.8.5.

4.5.5 Supporting names with crash-prone substrings

As mentioned earlier in Section 2.8.2, there are some error-prone substrings inside the entity names that can abort the extraction. The entity names with such strings are extracted. However, their children are not extracted, as it is not possible to use such substrings as parameters.

For determining whether a string contains an error-prone substring, the method `validateName` is used. The method is created inside the inner static class called ValidateUtils. The class is located inside the MetaDaoImpl. The method returns a name if the name is valid for sending as a parameter to narrow down the search for the extraction. Otherwise, returns `null`. If the string contains an error-prone substring, the `null` is used as the parameter instead of the ancestor's name in the methods of the DatabaseMetada class. Moreover, the name of the ancestor we have and the name of the ancestor retrieved using the Result Set are compared. It adopts the same idea as in Section 4.5.4.

The solution is present in Listing 4.4. Its table and schema names are validated. The `null` values or their names are used as arguments of the `getColumns` method. Then only columns that are located in the table and schema that are sent as arguments of the method `getTableColumns` are stored.

The same solution is applicable for procedures, functions, procedure columns, and function columns. Every time we want to retrieve children of a database entity, the names of all the ancestors should be validated using the method `validateName` and compared with the names

Code listing 4.4 The method for extracting columns of the table

```
private List<Column> getTableColumns(String catalog, String schema,
String table) {
    String tableNameAsPattern = ValidateUtils.validateName(table);
    String schemaNameAsPattern = ValidateUtils.validateName(schema);
    List<Column> columns = new ArrayList<>();
    try (Connection connection = dataSource.getConnection();
         ResultSet resultSet = connection.getMetaData()
         .getColumns(catalog, schemaNameAsPattern,
         tableNameAsPattern, null)) {
        while (resultSet.next()) {
            if (resultSet.getString(ColumnLabels.TABLE_SCHEMA)
            .equals(schema) &&
            resultSet.getString("TABLE_NAME").equals(table)) {
                Column column = new Column();
                column.setName(resultSet
                .getString(ColumnLabels.COLUMN_NAME));
                column.setDataType(resultSet
                .getInt(ColumnLabels.DATA_TYPE));
                column.setTypeName(resultSet
                .getString(ColumnLabels.TYPE_NAME));
                column.setCanBeNull(ConvertUtils
                .columnNullableIntToBool(resultSet
                .getInt("NULLABLE")));
                columns.add(column);
            }
        }
    } catch (SQLException e) {
        ...
    }
    return columns;
}
```

Code listing 4.5 Validation of the unknown procedure information.

```
@Override
public void writeProcedure(IResDataType dictSchema,
Procedure procedure) {
    ...
    List<RoutineColumn> unknownColumns = procedure
    .getUnknownColumns();
    if (!unknownColumns.isEmpty()) {
        writeProcedureWithoutParams(dictSchema, procedure);
        logger.log(Categories
                .extractionErrors()
                .routineColumnsAreUnknown()
                .columns(unknownColumns)
                .catalog(procedure.getCatalog())
                .schema(procedure.getSchema())
                .name(procedure.getName()));
        return;
    }
    if (procedure.getHasReturnValue() == null) {
        writeProcedureWithoutParams(dictSchema, procedure);
        logger.log(Categories
                .extractionErrors()
                .unknownProcedureReturnValue()
                .catalog(procedure.getCatalog())
                .schema(procedure.getSchema())
                .name(procedure.getName()));
        return;
    }
    ...
}
```

that are sent as parameters afterward.

It is worth mentioning that a catalog name cannot be validated because it cannot be compared after the validation. It is mentioned in Section 4.5.4.

4.5.6 Save routines with unknown information

There is some information about a procedure and a function that can be unknown. The absence of some information can give as a wrong entity saved at the end.

Procedure type information can be unknown. The information is stored in the "PROCEDURE_TYPE" attribute. In that situation, the best way is to save only the names of these procedures with unknown "PROCEDURE_TYPE" and log an error, but do not abort a program.

A function or procedure column can have an unknown type. The information is stored in the "COLUMN_TYPE" attribute. The decision is to save only the names of these routines where a column with the unknown type exists and log an error, but do not abort a program.

The implementation of solutions to the problems can be found in the method `writeProcedures` in the DictionaryWriterImpl class in Listing 4.5. If any unknown information is found, the method transfers responsibility for saving the procedure to the `writeProcedureWithoutParams` method. Similar approach is introduced in the `writeFunctions` method of the DictionaryWriterImpl class.

Code listing 4.6 The method for creating a return type of the routine.

```
private IResDataType getReturnType(Routine routine) {
    List<RoutineColumn> returnColumns = routine.getReturnColumns();
    if (returnColumns.size() == 1) {
        return dictionary.getBuiltinDataType(returnColumns.get(0)
        .getDataType());
    } else if (returnColumns.size() > 1) {
        IResDataType dictTableType = dictionary
        .createTableType(getDictName("ReturnTypeOf_"
        + routine.getName()), EntityProps.DB_DYNAMIC_RESULTSET_TYPE,
        SOURCE_TYPE, null);
        for (RoutineColumn column : returnColumns) {
            IResDataType columnDataType = dictionary
            .getBuiltinDataType(column.getDataType());
            dictionary.createColumn(getDictName(column.getName()),
            dictTableType, columnDataType, SOURCE_TYPE, null);
        }
        return dictTableType;
    }
    return null;
}
```

4.5.7 A function and a procedure with identical names

Functions with the same name or procedures with the same name located in the same schema are not saved due to the solution that is implemented and explained in Section 4.5.3. However, a function and a procedure with the same name and similar signature can be extracted due to an invalid extraction of the PostgreSQL database. In the PostgreSQL database, similar procedures are extracted for all functions. It means that all functions are procedures too. Nevertheless, sometimes the signatures of the procedure and function representing the same entity may vary. In PostgreSQL, functions having “out” parameters are extracted as functions with “out” parameters and as procedures where “out” parameters are extracted as columns of the return type.

A routine signature may vary depending on whether the routine is extracted as a function or as a procedure. Considering all the information above, the decision is to compare routines only by their names. If a procedure or a function with an identical name has been stored, the storage of the new routine is omitted.

A procedure is similar to a function, and it is not recommended to store the routine twice. Therefore, a verification of routines with similar signatures in the methods `writeProcedure`, `writeFunction`, `writeProcedureWithoutParams`, and `writeFunctionWithoutParams`is implemented.

4.5.8 Return type of the routine

The function or procedure can have multiple return value columns or multiple Result Set columns. If there is one such column, it is saved as a primitive return data type. If there is more than one return column, it is saved as a table data type, which saves every column of the return type to the table.

Listing 4.6 shows how the DictionaryWriterImpl class’s private method for defining the return type of the routine to save into the dictionary is implemented.

Code listing 4.7 SQL script for creating a function in an Oracle database

```
CREATE OR REPLACE FUNCTION MyFunction(input1 IN INTEGER,
input2 IN INTEGER, output1 OUT INTEGER, output2 OUT INTEGER)
RETURN INTEGER IS
BEGIN
    output1 := input1 + input2;
    output2 := input1 * input2;
    RETURN 1;
END;
```

4.5.9 Return types and out parameters interconnection

What can be extracted as a return type from the routine is limited by the DatabaseMetaData class. The limitations include that a one-column table type and a basic data type cannot be distinguished from one another, and we are unable to implement a record data type since the API does not give us any information about its columns. Information about how the record data type is extracted can be found in Section 2.7.14.

In PostgreSQL, it is possible to recreate a record data type using columns that express out parameters of a routine. However, it is not possible for Oracle. In Oracle, out parameters and return type are not interconnected. The example of the script in Listing 4.7 is valid for Oracle but not for PostgreSQL because of the difference between return type and out parameters.

4.6 Automated tests

In order to ensure the correct functionality of the JdbcDialect class and to verify that it saves only permitted entities, automated testing has been conducted.

Similarly, automated testing has been performed for the MetaDaoImpl class to ensure proper extraction and storage of all entities in DTOs for the PostgreSQL database. Additionally, catalog and schema extraction and filtering for four different DBMSs (Teradata, Oracle, PostgreSQL, and MSSQL) have been tested to ensure compatibility with various database structures.

The DictionaryWriterImpl class has also been subject to automated testing to verify the successful saving of DTOs.

The JdbcExtractorImpl class has undergone testing for the PostgreSQL database and, to a lesser extent, for the Teradata, Oracle, and MSSQL databases to ensure the complete scenario from extraction to saving is functional. Though the extent of testing was not as comprehensive for the non-PostgreSQL databases, the absence of errors was confirmed.

The automated tests encompass both integration and unit testing and rely on the Manta development environment. Changes made to the databases used for JDBC Scanner testing must be reflected in the automated tests. Code coverage is at 70%.

4.7 The JDBC Scanner performance with an unsupported DBMS

To test the JDBC Scanner with an unsupported DBMS, we chose to use MySQL. The purpose of this test is to evaluate Scanner's performance with untested dialects.

Testing with a MySQL database showed that the JDBC Scanner does not extract data properly due to incorrect extraction and filtering of databases.

MySQL has a two-segment database structure with no schemas, similar to Teradata. However, the implementation differs as Teradata extracts databases using the `getSchemas` method, while

MySQL extracts databases using the `getCatalogs` method of the DatabaseMetaData class.

In conclusion, the test of the JDBC Scanner with an unsupported DBMS such as MySQL has highlighted some limitations and issues in its performance. The improper extraction and filtering of data from MySQL database revealed the Scanner's lack of compatibility with certain dialects. This highlights the importance of thorough testing and validation of software components with various database management systems to ensure their proper functionality.

Chapter 5
Conclusion

The feasibility of implementing a generic extractor for RDBMSs has been demonstrated. The initial implementation has been completed and tested, demonstrating its ability to retrieve details about the database structure. The extractor has the ability to collect a range of information, including procedure, function, table, and view names, and catalog and schema details. Moreover, it offers insights into the parameters and return types in these routines, as well as the columns and its data types associated with the tables and views. Implementing the JDBC API presents various challenges due to the unique limitations and atypical behaviors that arise with different database management systems. These factors include variations in database structure, issues with incorrect extractions, and variations in the information that can be extracted for each DBMS.

The implementation of the JDBC metadata extractor adheres to Manta standards, and extracted entries are stored using the Manta dictionary. Additionally, the generic dictionary structure has been implemented.

Overall, the successful implementation and testing of the generic metadata extractor using the JDBC API, along with its potential benefits, make it a valuable contribution to the field of database analysis and data lineage. Further development and optimization of the extractor could lead to even more efficient and powerful database analysis.

Chapter 6
Future Work

The implementation of the JDBC metadata extractor provides important information about the structure of the database. However, there are many things to do in the future to enhance its capabilities and usability.

Better generalization of filtering and extracting of catalogs and schemas is one of the most crucial and urgent improvements that must be made in order to reduce the possibility of incorrect extraction of unsupported DBMS.

One of the improvements that can be done is better realization of the array data type. Currently, it is stored as an array without any information about the array elements' data type. Adding this information will provide a more complete understanding of the data type array in the database.

Another important aspect that needs to be addressed is the extraction of information about primary and foreign keys. This information is crucial for understanding the relationships between tables in the database and can be used for optimizing queries and improving performance.

In addition to primary and foreign keys, the extraction of more attributes of columns is also essential. This includes information such as default values, length, precision, and other relevant information.

Although the testing of the PostgreSQL database extraction is already implemented, it is essential to test the JDBC Connector more thoroughly. This includes testing classes with other dialects, especially the MetaDaoImpl class. Testing these dialects will ensure that the JDBC metadata extractor is robust and can handle different databases.

Another important aspect that needs to be addressed is the creation of the "manta-connector-jdbc-dictionary-mapping" module and relevant classes for storing information about the connection in a CSV file. This will provide an easy way to store and retrieve information about the database connection.

Finally, collecting the time of database calls and printing statistics at the end using the DbCallStatistics class of Manta can provide important insights into the performance of the extractor.

Bibliography

1. MCKENZIE, Cameron. *What is Java Database Connectivity (JDBC)?: Definition from TechTarget* [online]. TheServerSide.com, 2019 [visited on 2023-02-21]. Available from: `https://www.theserverside.com/definition/Java-Database-Connectivity-JDBC`.
2. *Connecting with DataSource objects* [online]. Oracle.com, [n.d.] [visited on 2023-03-05]. Available from: `https://docs.oracle.com/javase/tutorial/jdbc/basics/sqldatasources.html`.
3. DAVID-ENGEL. *Building the connection URL - JDBC driver for SQL server* [online]. Microsoft.com, [n.d.] [visited on 2023-02-18]. Available from: `https://learn.microsoft.com/en-us/sql/connect/jdbc/building-the-connection-url?view=sql-server-ver16`.
4. *Client-side Authentication* [online]. Oracle.com, [n.d.] [visited on 2023-02-20]. Available from: `https://docs.oracle.com/cd/A97335_02/apps.102/a83722/secure4.htm`.
5. *Initializing the driver* [online]. Postgresql.org, [n.d.] [visited on 2023-02-20]. Available from: `https://jdbc.postgresql.org/documentation/use/`.
6. *HOW TO: Connect to SSL enabled DB2 through the JDBC V2 connector in IICS* [online]. Informatica.com, [n.d.] [visited on 2023-02-20]. Available from: `https://knowledge.informatica.com/s/article/000176431?language=en_US`.
7. *SSL Connection to Oracle DB using JDBC, TLSv1.2, JKS or Oracle Wallets (12.2 and lower)* [online]. Oracle.com, [n.d.] [visited on 2023-02-20]. Available from: `https://blogs.oracle.com/developers/post/ssl-connection-to-oracle-db-using-jdbc-tlsv12-jks-or-oracle-wallets-122-and-lower`.
8. *Teradata JDBC Driver Reference* [online]. AmazonAws.com, [n.d.] [visited on 2023-02-20]. Available from: `https://teradata-docs.s3.amazonaws.com/doc/connectivity/jdbc/reference/current/frameset.html`.
9. *Interface DatabaseMetaData* [online]. Oracle.com, 2020 [visited on 2023-02-20]. Available from: `https://docs.oracle.com/javase/7/docs/api/java/sql/DatabaseMetaData.html`.
10. *Retrieving and modifying values from Result Sets* [online]. Oracle.com, [n.d.] [visited on 2023-02-24]. Available from: `https://docs.oracle.com/javase/tutorial/jdbc/basics/retrieving.html`.
11. *SQL Server Synonyms and PostgreSQL Views, Types, and Functions* [online]. Amazon.com, 1989 [visited on 2023-03-01]. Available from: `https://docs.aws.amazon.com/dms/latest/sql-server-to-aurora-postgresql-migration-playbook/chap-sql-server-aurora-pg.tsql.synonyms.html`.

12. *Types (java platform SE 8)* [online]. Oracle.com, [n.d.] [visited on 2023-03-12]. Available from: https://docs.oracle.com/javase/8/docs/api/java/sql/Types.html.

13. *Introduction - Apache Maven* [online]. Maven.org, [n.d.] [visited on 2023-02-15]. Available from: https://maven.apache.org/what-is-maven.html.

14. *Spring Framework Documentation* [online]. Spring.io, 2002 [visited on 2023-03-21]. Available from: https://docs.spring.io/spring-framework/docs/current/reference/html/.

15. TOUSEK, Jiri. *Data Dictionary (manta-connector-common-dictionary)* [online]. Manta, [n.d.] [visited on 2023-03-15]. Available from: https://manta-io.atlassian.net/wiki/spaces/MT/pages/1939243042.

The content of the attached media