CZECH TECHNICAL UNIVERSITY IN PRAGUE
FACULTY OF INFORMATION TECHNOLOGY

ASSIGNMENT OF BACHELOR'S THESIS

Title:	Tool for metadata extraction from database Netezza
Student:	Boris Laskov
Supervisor:	Ing. Michal Valenta, Ph.D.
Study Programme:	Informatics
Study Branch:	Software Engineering
Department:	Department of Software Engineering
Validity:	Until the end of summer semester 2017/18

Instructions

The aim of the thesis is to design and develop a metadata extraction tool for the Netezza database and integrate the tool into the Manta project.

1. Describe the structure of metadata in the Netezza database and compare it with metadata in the Oracle and MS SQL Server.
2. Prepare a list of metadata that is necessary to extract from this database for static analysis, verify its availability, and identify possible ways of its extraction from the database.
3. Based on the analysis design a tool for metadata extraction from the Netezza database.
4. Implement the prototype that extracts metadata into existing data structure in the Manta project. Use the appropriate architecture to allow integration of this tool into Manta. To access the database, use a suitable framework.
5. Create a set of test data and automated tests to verify that the implemented solution works correctly. Create user documentation for this tool.

References

Will be provided by the supervisor.

Ing. Michal Valenta, Ph.D.
Head of Department

prof. Ing. Pavel Tvrdík, CSc.
Dean

Prague December 22, 2016

Czech Technical University in Prague
Faculty of Information Technology
Department of Software Engineering

Bachelor's thesis

Tool for metadata extraction from database Netezza

Boris Laskov

Supervisor: Ing. Michal Valenta, Ph.D.

21st April 2017

Acknowledgements

I would like to thank my supervisor, Ing. Michal Valenta Ph.D., for his guidance and valuable advice. Also, I would like to thank the team behind Manta Flow for their help in understanding the context of this work and the structure of their software.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended. In accordance with Article 2373 of the Act No. 89/2012 Coll., the Civil Code, I hereby grant a nonexclusive authorization (license) to utilize this thesis, including any and all computer programs incorporated therein or attached thereto and all corresponding documentation (hereinafter collectively referred to as the "Work"), to any and all persons that wish to utilize the Work. Such persons are entitled to use the Work in any way (including for-profit purposes) that does not detract from its value. This authorization 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 Work, combine it with another work, and/or include the Work in a collective work.

In Prague on 21st April 2017 .....

Czech Technical University in Prague
Faculty of Information Technology

Citation of this thesis

Laskov, Boris. *Tool for metadata extraction from database Netezza.* Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2017.

Abstract

In order to perform static analysis of an SQL script we need to have the information about database objects, which this script works with. The objects and the information they can be described by may vary depending on the database.

The aim of this thesis is to explore the environment of Netezza database and collect knowledge about objects that can be found inside it, their descriptive data and ways to extract that data. The resultant theoretical material was tested on a practical example of designing and implementing a Netezza-specific extractor module for Manta Flow.

Manta Flow tool is used to statically analyze SQL code and visualize the structure of a database. Our application contributes to bringing the support of Netezza databases into it.

The result of the practical part of this thesis is a fully-functional extractor. Although it does not analyze SQL code by itself, it provides required data for future processing by other modules. The theoretical part can be useful for understanding data structures in Netezza.

Keywords Netezza, metadata, database, Java, automatic metadata extraction

Abstrakt

Pro statickou analýzu SQL skriptů je třeba získat detailní informaci o objektech, které jsou ve skriptech odkazované. Této informaci říkáme metadata. Struktura metadat se v různých databázích liší.

Cílem této práce je prozkoumat databázový stroj Netezza, popsat strukturu informací o objektech (metadata) a zjistit, jak je možné je získat. Postup je prakticky ověřen návrhem a implementací metadata extraktoru databázového stroje Netezza pro nástroj Manta Flow.

Manta Flow slouží ke statické analýze SQL kódu a vizualizaci struktury databází a datových toků. Aplikace, která je výsledkem této práce, tedy plně funkční extraktor metadat pro db. stroj Netezza, významně přispívá k podpoře tohoto db. stroje v nástroji Manta Flow. Teoretická část práce je užitečná pro pochopení struktury metadat db. stroje Netezza.

Klíčová slova Netezza, metadata, databáze, Java, automatická extrakce metadat

Contents

List of Figures

List of Tables

Introduction

Nowadays, almost every company uses one or more databases to store various documents, information about their sales, customers and other relevant records. Instead of keeping such things written on paper, it is far more convenient to have a computer-based solution that is capable of storing practically all kinds of data. Moreover, databases ensure its safety and high availability. They enable companies to automate many routine processes thus saving much time and human resources. Relatively low costs needed to create and integrate a database and, more importantly, potential expenses reduction for the business make such storages ubiquitous [1].

However, the longer the company exists, and the bigger it grows, the harder it becomes to maintain its databases. Due to the continuously increasing amount of data they store and of people they are used by, if not designed properly, they start to perform slower, taking more time to process even simple transactions. So it becomes necessary to update their structure to keep up with corporate standards. New services offered by the company also imply not only creating new databases but also changing and extending existing ones. In addition, while such interventions in their architecture are being performed, old records must stay intact, because many pieces of information saved in databases often belong to the most valuable things the company has.

Updating the structure of a large and complex database can be difficult to do and can take a long time. It is also important to have a clear picture of what and how is stored inside in order not to lose any data and be sure that it is kept and managed in a proper way.

Such problems can be made much easier with a good software solution, which can connect to a database, retrieve the information about its inner structure, analyze it and provide a clear visualization of it. The purpose of this thesis is to extend one such tool – Manta Flow – by adding a new module to it that will bring the support for extracting the descriptive data (metadata) from Netezza database. The subsequent analysis will be performed by other modules and is not a part of this project.

Chapter description

Chapter 1 will contain more details about the aim of the thesis; it will define it more precisely.

Chapter 2 will give you an overview of Netezza database and explain the objects that can be found in it. Also, we will compare them to the ones from Oracle Database and Microsoft SQL Server.

In Chapter 3 we will describe the metadata to extract and the ways to accomplish it. Then there will be a list of requirements for our extractor and its main use case.

Chapter 4 will provide the information on technologies used to create our application and its structure (with a detailed overview of how it works).

In Chapter 5 we will talk about different automated tests we have and the data they are supplied with.

Chapter 6 will show step-by-step instructions for integrating our extractor into another project and launching it.

Finally, we will draw conclusions of this thesis.

Chapter 1

The aim of the thesis

In this chapter, we will give more details about the objectives of the project and its desired results. Firstly, Section 1.1 will clarify the conception and purpose of Manta Flow tool. Then, in Section 1.2 we will define the aim of this project based on the previous context. Section 1.3 will give you the picture of the current state of solutions for the problem. Finally, in Section 1.4 we will outline the main topics for the theoretical and practical parts of this thesis.

1.1 Manta Flow

Manta Flow is a commercial software tool which is designed to visualize database objects and relations between them. With the help of it, administrators can always have a well arranged graphical representation of the architecture they are supporting and working with, allowing them to maintain and update it faster and with fewer mistakes. Manta Flow contains modules called extractors that are used to pull out the structure of databases – there is no need to provide the scripts they were constructed with manually.

1.2 Purpose of this work

Database systems from various vendors differ in many ways. Each of them has its own way of how it represents, keeps and manages information inside. That is why Manta Flow has separate extractors – each of them is responsible for dealing with a specific database. The aim of this thesis is to describe the metadata of Netezza databases, find possible ways to extract it, and then make use of this knowledge by designing and implementing a prototypical solution – *an extractor*, which should meet the requirements of Manta Flow software so it can be seamlessly integrated with it.

1.3 Current state

According to the official documentation, there are several ways to get information about objects that are defined in the database. They differ in commands one needs to execute and in privileges one requires to be able to fully retrieve the relevant info. We need to examine them and make a comparison, so later we can choose the best one for our extractor.

As for Manta Flow, for now, it does not have any extractor for Netezza. Nevertheless, it has many extractors for other databases systems. It also has an interface that every new extractor must implement to become a part of Manta Flow. It should also work with some predefined structures of this project, for example, it must be able to fill its dictionary. We will describe these technical aspects in more detail in the following chapters.

1.4 Theoretical and practical parts

The theoretical part of this thesis will be mainly focused on the describing the kinds of data structures that can be found in a Netezza database. They will also be compared to those from Oracle Database and Microsoft SQL Server. Then, we will discuss the ways of retrieving their metadata and user privileges needed for it. In the end, we will design a list of requirements and a use case for our extractor.

The practical part will consist of details about the implementation of our module. We will give details about what technologies we use, which parts it has and how it works. We will also provide the information on the automated tests we will create. Finally, there will be a manual about how to use our extractor in another project.

Chapter 2

Analysis

In this chapter, we will explore the environment of Netezza database and gather the knowledge to understand and be able to describe objects stored inside it.

Firstly, in Section 2.1, we will define a few terms, which will help us in describing objects of a database. In Section 2.2 we will say something about what Netezza actually is. Then, in Section 2.3 we will see what objects are there in Netezza and what info we can describe them with. Sections 2.4 and 2.5 will provide a comparison of metadata found in Netezza with that in Oracle and Microsoft databases.

2.1 Definition of terms

Data definition language (or DDL) is a special category of Structured Query Language (SQL) commands. It contains commands for defining database objects, for example, tables, views and others [2].

Data manipulation language (or DML) is another category of SQL that refers to commands for handling data stored in objects which were defined by DDL queries. DML commands can be divided into two main categories:

- The first one provides a way to actually manipulate data – add, modify, and remove it. It contains self-explaining `INSERT`, `UPDATE` and `DELETE` commands.
- The other category enables users to access stored information. In consists of only one command – `SELECT`. With the help of it, it is possible to both output info as is and use conditions, functions and other database structures to change the final result set [2].

DDL scripts (or create scripts) are files that contain commands written in data definition language. Each of them can be responsible for defining one

or more database objects. Since the majority of the commands that construct new objects starts with the keyword `CREATE`, such files are often referred to as create scripts. Using them as a source of input in the database shell eliminates the necessity of manually typing `CREATE` commands.

Metadata "is data that describes other data" [3]. In our case, by metadata we mean information that helps us to learn what objects are in the database and how they were created. When we do not know what the database structure looks like, its metadata becomes crucial to understanding and visualizing it.

2.2 Netezza overview

Netezza is an American company (acquired by IBM in 2010), the developer of high-performance data warehouse appliances, which are focused on massively parallel processing (MPP). It creates solutions that are designed to store and process petabytes of information. Their distinctive feature is the usage of FPGAs along with Intel-based CPUs on the nodes where data is processed.

The architecture of appliances consists of two main levels:

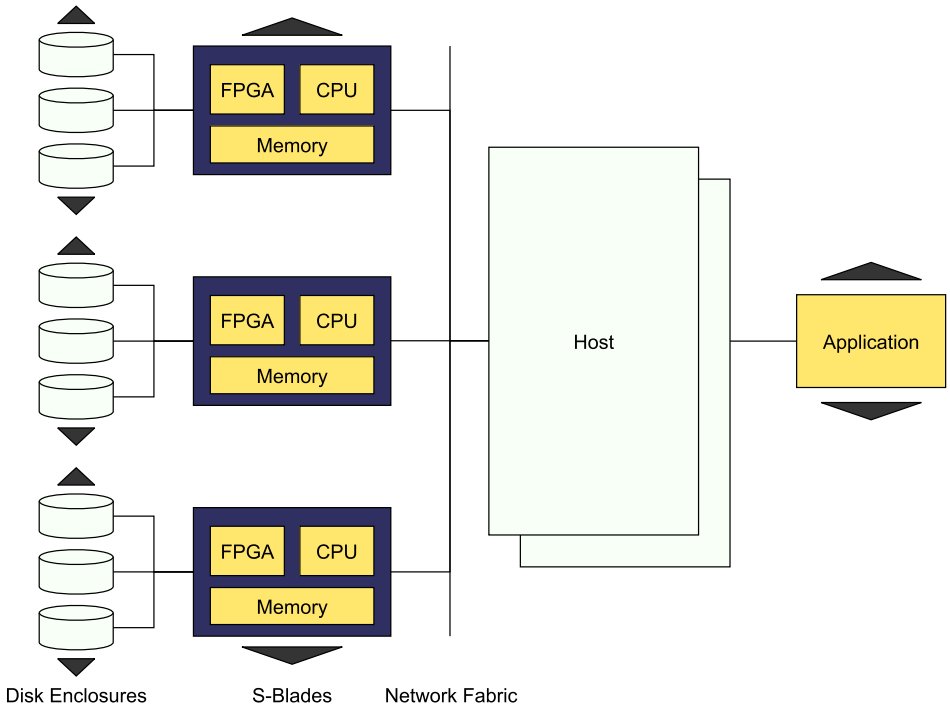

Figure 2.1: Netezza Architecture

- The first level is called the host and is represented by an SMP server. It receives queries from the client using it and performs the preprocessing – it creates an optimized execution plan. It breaks each task into subtasks (snippets), which are then effectively distributed across S-Blades from the second tier for parallel execution.
- The second level consists of one or more snippet blades (S-Blades). They are powerful independent servers with lots of RAM, high-performance CPUs, and FPGAs, which receive subtasks from the host and do the actual computation on the information, which is stored on one or more hard disks the S-Blade is connected to. On this machines takes place the actual data processing [4].

The detailed description of the Netezza appliances goes beyond the scope of this thesis. The key point is that they use a proprietary database. The analysis and extraction of its metadata are the purposes of this project. From now on this database will often be referred to simply as "Netezza".

2.3 Metadata structure

Please note that this whole section is mostly based on the info from the following sources: [5], [6], [7] and [8].

Also note that it is not a full, definitive guide for metadata in Netezza. Neither every object nor every object's parameter is listed here. As we will see in future sections, for designing and implementing an extractor, we will not need that amount of information (due to requirements of Manta Flow – what it demands and what it can make use of). However, all substantial parts are in place.

2.3.1 Objects in Netezza

On the root level, there are databases (DBs). Each database by default has one schema. Since release 7.0.3 of Netezza Platform Software (NPS), we can configure the system to use multiple schemas. The user can switch between databases with the `SET CATALOG` command and between schemas in the current database with the `SET SCHEMA` command.

Schemas contain actual database data structures. They are:

Tables
Classical relational database tables, which consist of columns. Every column has a name and a data type. Tables can have constraints for specifying rules in them (for example, `PRIMARY KEY` (PK), `FOREIGN KEY` (FK), etc.) [9], but in Netezza, they are not checked. That is why it is user's responsibility to ensure that they will not be violated.

From the metadata point of view, a table can be described by its name, table-level constraints (e.g., PK, FK, etc.) and a list of columns. Each column has a name, a data type and, sometimes, column-level constraints (e.g., `NOT NULL`, `UNIQUE`, etc.). Neither table-level nor column-level constraints are checked. Also, there are some additional parameters:

- `DISTRIBUTE ON` stands for one to four specific columns (its default value is `RANDOM`) based on which the data is split across different processing nodes (and their hard drives) [10].
- `ORGANIZE ON` represents one to four columns, values of which are used to group records within the table.
- `ROW SECURITY` indicates that this table is secure on row-level – user must have special permissions to view or change some rows.

Derived tables
The lifecycle of a derived table matches the lifecycle of an SQL query, during which it is created. As soon as the request is completed, the table is removed [11]. They can be used as intermediate containers that have a shape of a regular table.

Temporary tables
Temporary tables behave like derived tables, but they are local to the current session rather than for a single request. The name of such table can overlap with the name of a regular, persistent one – in that case, the latter will be unavailable until the former exists. The temporary table will be accessed by default. However, it can be dropped by the user before the session expires [11].

External tables
This kind of tables is stored outside of a database. They can be used for loading data into the database and unloading it to files on disk. Another use case of keeping data external is the ability to move it between different systems or to simply back it up.

External tables do not have an `ORGANIZE ON` parameter, but they do have many additional ones. For example, `DATAOBJECT` shows where the table is saved in the file system. There are many more parameters, for instance, responsible for handling the format of how some data types are stored, specifying delimiters between columns, and so on.

Views
There are two types of views in Netezza database: regular and materialized.

- Regular views are simple structures that are not created physically. Netezza only generates a query rewrite to enable `SELECT` commands on them. If we add a column to the table the view is based on, it will not appear in the view. If we delete a column that was accessible through the view, future queries will end with an error.

- Materialized views are physically created (stored in a unique table). `SELECT` statements on them can show better performance because often there are fewer records in views, and this data is sorted on materialized view creation. Only one base table can be used as a source. Although making changes straight on views is not allowed, it is possible to do them on the base table, and they will propagate to the view. To keep data sorted, it is necessary sometimes to refresh materialized views manually or to set an appropriate threshold, which defines how many percents of unsorted records can be present until the view is automatically refreshed (default value is 20).

A very important descriptive parameter of a view is the query, with the help of which it is generated from its base table.

Sequences
A sequence is an object based on the integer data type which has minimum and maximum values, a starting value and a step by which the value is increased (once returned). Sequences have the `GET NEXT VALUE` method, which returns a number from the defined interval. In the majority of cases, it is supposed to be unique, but if we set the `CYCLE` property, once we move up to the maximum value, we will start getting numbers from the beginning of the interval again. Values returned by a sequence can be used, for example, for generating artificial primary keys – special values to uniquely identify rows, which are not a logical part of the rest of the data stored in this table and should be created aside.

Stored procedures
Instead of building all logic outside of a database (in external applications, which are connected to it), sometimes it is a better choice to implement parts of it straight in Netezza. It can save us time because for processing information in place there is no need to transfer it between systems. For such purposes, Netezza has special objects called stored procedures. They enable users to define functionality and save it as a database object.

They are written in NZPLSQL language, can take input values (arguments) and supports return values. Their arguments can be either a predefined list of built-in primitive data types or a special `VARARGS` parameter for a varying number (0 to 64) of inputs of different types. Their return value can be not only a primitive data type but also a `REFTABLE(<name_of_table>)`. In that case, the procedure returns a result set, which has the same structure as the specified table. This table must be created before the procedure and cannot be deleted while it exists.

Procedures can also be obfuscated. It means that once created and saved, their body (actual code of what they do) appears not as NZPLSQL code (like it was written), but as a sequence of characters that looks random and does not mean anything to the person reviewing it. Hiding the original body of a procedure can be done for legal reasons if it contains some information its

author does not want to be stored or viewed as plain text. In Netezza, procedures can be overloaded. It means that there can exist multiple procedures with the same name, but with different arguments. When called, an appropriate one will be picked and executed (considering the arguments caller provided).

As for metadata, the most important parts are the name, arguments, return type and the body of the procedure.

User-defined functions (UDXs)
This is another type of executable objects. Like procedures, functions can be overloaded; they have arguments and return values. But the code of UDXs is written in C++. It is firstly compiled with the `nzudxcompile` tool and registered in the system with an appropriate `CREATE` SQL command. Functions can then be invoked in queries to compute something based on supplied values. They can be of several types:

- User-defined aggregates (UDAs): This is an aggregate function – it calculates a single return value based on the given column.
- User-defined functions (UDFs): This is a more generic function, which can take zero or more arguments. Like a UDA, it is scalar – it produces only one return value.
- User-defined table functions (UDTFs): These functions return a table-like object rather than a single value of a primitive built-in data type. They can be called in a `FROM` clause of a statement. UDTFs help the user, for instance, to rewrite input columns in a more verbose form or instead summarize their info and present it in a more elegant way.
- User-defined shared libraries: Libraries enable the user to use the same C++ code in different UDXs in a clean and simple way. There is no need to copy and paste it to every function that uses it.

Metadata of UDXs include their names, arguments and return values. However, it appears to be impossible to extract their bodies because they are written in C++ and are stored in a compiled form. Here are more parameters that describe UDXs:

- `[NOT] FENCED` indicates if a function is executed in its own process with protected address space (this mode is called fenced) or not. Fenced mode negatively affects performance, but in the case of poorly designed UDXs, it helps the system to avoid crashes.
- `[NOT] DETERMINISTIC` shows if a function is deterministic or not. If it is, for the same arguments it must always return the same value.
- `MAXIMUM MEMORY` represents the possible amount of memory that this function can use. It can be written as a number together with a letter, which stands for the units used ('b' for bytes, 'k' for kilobytes, 'm' for megabytes or 'g' for gigabytes).

- `TABLE [ [, TABLE ] FINAL ] ALLOWED` represents invocation options of a UDTF. For example, if `TABLE FINAL ALLOWED` is specified, and the user calls it with `TABLE WITH FINAL(udtf())` command, then this function will be invoked once more after all input values are processed.
- `LOGMASK` specifies the level of logging applied to the function.
- `STATE` defines data types of state variables. This is a UDA-only parameter. Aggregates use these variables to save their state externally while running.
- `[NO] DEPENDENCIES` shows if the UDX is dependent on one or more shared libraries.
- `EXTERNAL CLASS NAME` specifies the name of the main class in C++ (the entry point of the function).
- `EXTERNAL HOST OBJECT` and `EXTERNAL SPU OBJECT` point to the corresponding files with compiled sources created by the `nzudxcompile` command.

Synonyms
Synonyms act as "nicknames" for other objects. They can be used to reference objects in different schemas and databases. Interestingly, when the user creates a synonym, NPS does not check if such object exists at all. It is also possible to create a single synonym for an overloaded procedure or function and then use it just like the original executable's name calling appropriate target object depending on the supplied arguments. One synonym can reference another one.

Synonyms can be described by the name and by the information about their target objects. However, this info contains only the location (the name of the database and the schema) and the name of the object. Its data type, for example, is not provided. If we would like to find it out, we will have to search for this object explicitly. Moreover, there are two main namespaces in Netezza: the first one keeps names of tables, views and sequences, and the second one – names of procedures, functions, and aggregates. Considering the fact that a synonym only knows the whereabouts and the name of its target, but not the target object itself, if we have, for instance, a sequence and a procedure with the same name, it is possible to create such a "nickname" for both of them. It will be stored as a single object, and its target will be resolved based on the syntax of the invocation command.

2.3.2 Fully qualified names

All objects have complete versions of their names called fully qualified names. It has three levels separated by dots:

- The first level denotes the database the object is located in.

- The second level represents the schema.
- The third level is the actual name of the object.

Let's imagine we have a table `CUSTOMERS` in `SALES` schema of `STORE` database. If we are inside that schema, we can refer to it simply as `CUSTOMERS`. If we are in that database, but in a different schema, we can access our table by the `SALES.CUSTOMERS` two-level qualified name. Finally, if we are in another database, we can still reach our table with its fully qualified name – `STORE.SALES.CUSTOMERS`.

2.3.3 SQL identifiers

Netezza supports two kinds of identifiers:

- Regular: they are case-insensitive, transformed to uppercase or lowercase depending on the global system settings. They cannot match words reserved by Netezza, they must start with a letter and contain letters, digits, underscores and a few other characters.
- Delimited: such identifiers are written in quotation marks. Unlike regular ones, they are case-sensitive. They can be constructed with the same symbols, but also the can have inner spaces (not leading or trailing, those are truncated) and special characters.

It is worth noting that it is possible to take a regular identifier in the appropriate system case and use it like a delimited one surrounding it with quotation marks.

2.4 Comparison with Oracle SQL

Oracle Database is an object-relational database management system (ORDBMS) created by Oracle Corporation. It was one of the first commercially available solutions of its kind. Unlike many others, it has support for user-defined objects.

Oracle database has a few objects that Netezza does not. When creating an extractor for it, it is necessary to correctly deal with a larger amount of metadata of even more kinds. Objects that are present in both database systems can differ too.

Information in this chapter is based on this source: [12].

2.4.1 Objects

Just like in Netezza, in Oracle we have databases, which contain schemas, which contain objects. There are two notable differences though:

- Firstly, a schema in Oracle is essentially a set of database structures owned by the particular user. It is not a simple container for objects; it is designed to be more like a user's workspace.
- Secondly, to switch to a different database one should use a special link to it – so it is not as easy as Netezza's `SET CATALOG` command.

Triggers
In contrast to Netezza, Oracle does have these objects, which are common in many other databases, called triggers. They are a special kind of stored procedures that, instead of being launched explicitly by the user, are fired automatically by the DB whenever a certain event occurs (so-called *triggering event*). Triggers can be useful, for instance, to calculate some derived values or refresh aggregation results based on issued `INSERT`, `UPDATE` or `DELETE` commands, or to control and modify transactions. There are three types of events a trigger can respond to:

- DML statements: triggers operate on the table or view level.
- DDL statements: triggers react to changes in the DB structure, for instance, to other objects being created, modified or deleted.
- Database: among triggering events there are global database operations like `STARTUP`, `SHUTDOWN`, `LOGON` and `LOGOFF`.

From the metadata point of view, triggers can be described by their definition and body. The definition consists of two parts:

- The *triggering statement* denotes the event and the object it is called on.
- The *trigger restriction* provides an additional condition that has to be true for the trigger to be executed.

The body is written in PL/SQL (PL here stands for Procedural Language) – Oracle's procedural extension for SQL. It contains the actual code that is executed once the trigger is fired.

Cursors
A cursor is another SQL concept that is missing in Netezza but is present in Oracle. It is a special structure through which the user can gain access to the information of DML statements being processed. Strictly speaking, cursors are not database objects – they cannot be declared independently. But they can be passed from one procedure to another as input parameters. That is why it is necessary to be able to distinguish them from other inputs (such as primitive data types) when parsing and processing metadata.

Functions and procedures
Executable objects in Oracle and Netezza have a few differences listed in Table 2.1.

Table 2.1: Comparison of executables in Oracle and Netezza

Netezza	Oracle SQL
Functions are written in C++.	Both functions and procedures are written in PL/SQL.
Procedures are written in NZPLSQL.	
Procedures can return one value or result set with a `RETURN` statement.	Procedures return multiple values with `OUT` and `IN OUT` parameters.

Packages
As opposed to Netezza, Oracle provides a convenient approach to putting together related procedures, functions, variables, cursors and some other objects. It is achieved with the data structure called a *package*. It has two major parts:

- The *package specification* contains declarations of objects that can be accessed from outside of the package.
- The *package body* has the actual queries of cursors and code of procedures and functions declared in the package specification. It can also contain declarations and definitions of objects that are not supposed to be public – that way they cannot be referenced from the outside. The body sometimes is not needed – for instance, when we want to have only public constants. They do not need a body and can be fully defined in the specification part.

As for metadata, the two parts listed above can be used to describe a package and its contents.

User-defined types
Unlike Netezza, Oracle supports not only built-in data types but user-defined ones as well. Such type can have the following components:

- A *name*, which uniquely defines that type in a schema.
- *Attributes*, which can be of both built-in primitive types and other user-defined ones.
- *Methods* that are presented by procedures and functions written in PL/SQL. They should be declared in the main `CREATE TYPE` query, but their implementation can be moved away, to the `CREATE TYPE BODY` part. This separation of declaration and implementation is similar to specification and body parts of a package.

Indexes
An index is a structure that can possibly speed up retrieving data from a table. It can be described by the table it is associated with and by one or more columns it is built on. Indexes are not compulsory.

Netezza does not support indexes, but similar functionality can be partially achieved with a materialized view (because data in it is sorted and saved separately from the table it is based on).

Database links
In order to access another database, the user must create a *link* to it. This link is a special object that has info such as the location of other DB and login credentials. Links are defined inside schemas with `CREATE DATABASE LINK` commands.

2.4.2 Fully qualified names

The full version of the name of an object in Oracle is `SCHEMA.OBJECT@LINK`. So if we take our previous example from Netezza (we had a table `CUSTOMERS` in `SALES` schema of `STORE` database), in Oracle referencing this table will look like `SALES.CUSTOMERS@STORELINK`, where `STORELINK` is a database link that points to the `STORE` database (if it is not the DB we are currently in).

2.4.3 SQL identifiers

Just like in Netezza, identifiers in Oracle can be of two types: here they are called quoted and nonquoted. The key differences between them are similar to those between Netezza's regular and delimited ones: nonquoted identifiers allow fewer character classes than quoted ones, but latter must be escaped with quotation marks.

- Nonquoted identifiers: case-insensitive, interpreted as uppercase, must start with a letter, can include alphanumeric characters, dollar and pound signs, cannot be reserved words.
- Quoted identifiers: case-insensitive, can begin with any character, can include any characters and symbols (except for double quotation marks and the null character), can match reserved words.

2.5 Comparison with Microsoft SQL Server

Microsoft SQL Server is another relational database management system solution. When compared to Netezza, it has a few more data structures (like Oracle does). Information in this chapter is based on these sources: [13], [14] and [15].

2.5.1 Objects

Like in both Netezza and Oracle, in SQL Server there are objects, which are inside schemas, which are inside databases. To switch between DBs located

on the same host, we can take advantage of the `USE` command (providing it with the name of the database we want to connect to).

Triggers
Unlike Netezza, SQL Server supports defining and using triggers. This concept is the same as in Oracle; however, a few differences take place. Some of them are listed in Table 2.2.

Table 2.2: Comparison of triggers in Oracle and Microsoft SQL Server

Oracle	SQL Server
Regular DML triggers can be fired before or after a specific command.	Only `AFTER` behavior is available.
Recursive calls is a commonplace.	Recursion is supported but must be enabled explicitly.
Per-statement and per-row triggers.	Only per-statement, no analog for Oracle's `FOR EACH ROW`.
`BEFORE` triggers have access only to the initial image of the table, `AFTER` triggers – only to the modified one. Both initial and modified images are available only in `FOR EACH ROW` triggers and only in the context of one row.	Both `BEFORE` and `AFTER` images of the table are available to any trigger.

Cursors
Cursors, which are missing in Netezza, are here as well. Akin to the Oracle ones, they can be passed between procedures. The concept is roughly the same, but in details, there are some differences. For example, cursors in Oracle are always local to the context they are defined in, whereas in SQL Server they can be global; Microsoft's cursors do not support parameters; etc.

Functions and procedures
In contrast to Netezza, both functions and stored procedures in SQL Server are written with Transact-SQL (T-SQL) – Microsoft's proprietary extension of SQL. Functions can be of two types:

- Scalar, which return exactly one value.
- Table-valued, which return a result set and can be used in `SELECT` statements just like a table or a view.

Stored procedures can return multiple values through `OUTPUT` parameters, while in Netezza they provide only one result with `RETURN` statement and do not support any but `INPUT` parameters.

In both executable types in SQL Server parameters can be `DEFAULT` (with a value assigned during creation) or `OPTIONAL` (not compulsory for launching a function or procedure).

User-defined types
Another feature that is not implemented in Netezza. Unlike Oracle's user-defined data types, Microsoft's one cannot be considered objects – they are simple aliases to the system types that already exist in the database with optional `DEFAULT`, `NOT NULL` or `CHECK` constraints [16].

Indexes
SQL Server allows defining indexes on tables and views to speed up data retrieving from them. In Netezza, there are no indexes, but their functionality can be emulated with a materialized view, which keeps its data sorted.

Linked servers
To access SQL Server databases located on remote machines or to work with other than SQL Server data sources, user must define them as *linked servers*. Microsoft's OLE DB API is used here. It offers a uniform data access between various sources.

2.5.2 Fully qualified names

In SQL Server a fully qualified name consists of 4 parts: the name of the server, the name of the database, the name of the schema and the name of the object. For instance, it can look like `MAINSERVER.STORE.SALES.CUSTOMERS` (taking our previous example and extending it with a link to another server called `MAINSERVER`).

2.5.3 SQL identifiers

Following Netezza's and Oracle's path, Microsoft offers two types of identifiers: regular and delimited. Considering the fact that we have already described them in previous sections, it is a little bit pointless in going through this info again. The keypoint is that we can use a wider variety of characters in delimited ones, but in that case, we must add quotation marks or, in SQL Server, square bracket around them. The complete set of rules goes beyond the scope of this work; it can always be found in the official documentation.

Chapter 3

Design

In the first part of this chapter, we will describe which metadata we need to extract and choose a way to do so. In the second part, we will provide some formal details on the specification of the application. Section 3.1 will list all the data that is supposed to be pulled out. In Section 3.2 we will discuss the ways to accomplish it and choose the most suitable one. Section 3.3 will provide the information on privileges that we need to perform the extraction. In Section 3.4 we will put together functional and non-functional requirements the application should meet. In Section 3.5 there will be a use case of our extractor.

3.1 List of necessary metadata

3.1.1 Extracted metadata

According to the requirements and capabilities of Manta Flow tool, the metadata of the following objects should be extracted:

- Database
  - Name
  - List of schemas inside it
- Schema
  - Name
  - List of objects inside it
- Table
  - Name
  - Columns:
    - Name
    - Data type

For external tables it will be nice to have the names of the files they are stored in and paths to them.

- View
  - Name
  - Columns:
    - Name
    - Data type
  - Definition
  - Type (regular or materialized one)

  For generating create scripts, it is necessary to know the definition of the view – the query which it was constructed with. To save a view for Manta Flow internally, we must have the names and the data types of its columns. In case the list of columns will not be present, we can parse the definition to find out these attributes.
- Sequence
  - Name
  - Data type

  For now, simply knowing about the sequence will be enough. Later, its data type may also be useful.
- Procedure
  - Name
  - Arguments
  - Return type
  - Body
- Function
  - Name
  - Arguments
  - Return type

  Due to the fact that user-defined functions in Netezza use compiled C++ sources, it is impossible to retrieve the code of their bodies. Nevertheless, we can still get their declarations to keep a record of existing functions.
- Synonym
  - Name
  - Location of the target object
  - Target object itself

  In order to create DDL scripts, we must know the database, the schema and the name of the object our synonym is pointing to. To save the synonym in Manta Flow, we must have the whole target object already saved in its dictionary.

3.1.2 Ignored metadata

The following metadata is *not* supposed to be extracted:

- Constraints
- User-defined shared libraries

3.2 Ways to extract metadata

There are several ways of how to get the information about objects that already exist in the Netezza database. Information in this chapter is based on the source [17].

3.2.1 The nzdumpschema command

The `nzdumpschema` command generates create scripts for all objects in a given database. It works only with object definitions – no data is extracted or saved. Here is an example:

```
nzdumpschema -R STORE nzdumpschema_result.sh
```

It outputs one shell script to the current folder. It contains DDL commands to recreate the whole `STORE` database from the start. This would be an ideal choice for us, but there are some limitations:

- Firstly, it requires `ADMIN` privilege. In practice, it is highly unlikely that a third-party software will be given such permissions (let's not forget that Manta Flow is a commercial product used by various clients, and databases it works with contain very precious data that can be easily destroyed by someone with such level of access).
- Secondly, according to the official documentation, it uses so much memory that the server can potentially run out of it and crash. This is inacceptable in production environment.
- Thirdly, in the documentation it is also stated that `nzdumpschema` is not supposed to be used regularly. It is designed to find and correct possible bugs in the DB.

That is why we cannot use it to get metadata for our extractor. Thankfully, there are some other options.

3.2.2 The nzbackup command

The `nzbackup` command is used to back up the entire database. It can save both definitions of objects and their data. For instance:

```
nzbackup -db STORE -dir /export/home/nz/nz_backup \
         -u flowuser -pw pswd -noData
```

The line above issues a backup command with credentials of the user `flowuser`. It backs up `STORE` database to the `nz_backup` folder. The `-noData` parameter states that the actual data should not be copied, so as a result we will have only object definitions (they are stored in XML files). This particular command does everything we need, but in our case we cannot use it either, here is why:

- Just like the `nzdumpschema`, the `nzbackup` is memory-consuming, but here instead of a crash and reboot we can run into a "memory limitation" error.
- The command requires a `BACKUP` privilege. Even if a user does not have appropriate permissions to see some objects in the DB, with this right he/she can back up these objects together with their data, even if `-noData` option is specified. That is why this privilege might not be granted to us.

So we have to move to another way of extracting metadata, which, by the way, is common in other extractors used by Manta Flow.

3.2.3 System views and tables

Netezza database has a set of system tables and views that are designed to provide information about existing objects and their properties. Those views that are most useful to us are listed below:

`_v_database`
All the databases present in the system.

`_v_schema`
All schemas in the current database.

`_v_table`
Names of all tables. It also has a column with the name of the schema the table is located into, so by adding a `WHERE` clause to the `SELECT` statement we can extract only the schema we want.

`_v_extobject`
Info specific to external tables. It can be joined with `_v_table` on `OBJID` column. We will extract `EXTOBJNAME` value that specifies a full path to the file on disk where our external table is saved.

_v_view
Names of all regular and materialized views. Their definitions are also here. Even if a view is created with a `SELECT * FROM` query, Netezza will rewrite it with a full list of columns, which, as the author tested, will not be truncated even if there are many of them. Different view types can be distinguished by specifying an appropriate `OBJTYPE` in `WHERE` clause.

_v_relation_column
Names and data types of all columns of tables and views. It can be joined on `OBJID` with `_v_table` or with `_v_view`.

_v_sequence
Names of all sequences in the current datatabse.

_vt_sequence
Additional information about sequences (such as data type, minimum and maximum values, etc.). It can be joined with `_v_sequence` on `SEQ_ID` (which is really an `OBJID`). Unlike all other sources listed here, it is not a system view but a management table.

_v_procedure
All information about stored procedures, including names, arguments, return values and bodies. To tell a procedure with a visible body from an obfuscated one, we will have to take a look at the first word of the body, which, according to the syntax, should be either `DECLARE` or `BEGIN`. If, instead, it is something meaningless, then it is obfuscated. However, it will not stop us from making create scripts, because supplying an obfuscated version of the body is a valid option from the DDL point of view.

_v_function
Names, argument lists and return types of UDFs and UDTFs.

_v_aggregate
Names, argument lists and return types of UDAs.

_v_synonym
Names of synonyms and locations of their target objects.

Views from the previous list give information about objects that are in all schemas in the current database. If we want to extract metadata from one particular schema, we can add an appropriate `WHERE` clause, because every view has a column the value of which represents the schema the object belongs to.

It appears that every object has an ID, which is unique in the database. When we need to join two or more views together to get complete info about some DB structure, we can do this by equating the columns with IDs. Almost everywhere they are called `OBJID`, only in `_vt_sequence` it is `SEQ_ID`. As far as the author has tested, it is just another name for a DB-global object ID, and the `JOIN`s are working correctly.

Note, that the task is to extract only user-defined items. The extractor should not pull anything predefined or built-in. To achieve this, we have to introduce additional restrictions. For example, when selecting table-like structures, we can add the following `WHERE` clauses:

- `WHERE UPPER(OBJTYPE) = 'TABLE'` when retrieving tables
- `WHERE UPPER(OBJTYPE) = 'EXTERNAL TABLE'` for external tables
- `WHERE UPPER(OBJTYPE) = 'VIEW'` for regular views
- `WHERE UPPER(OBJTYPE) = 'MATERIALIZED VIEW'` for materialized views
- `WHERE BUILTIN = false` for UDXs and procedures

This way we will only get user objects, not system ones. There is still a problem: metadata of databases and schemas does not have anything that would help telling if they are defined by a user or by the system. It was found experimentally, that there is one built-in DB called `SYSTEM` and two predefined schemas in every database: `DEFINITION_SCHEMA` and `INFORMATION_SCHEMA`. They are special system objects that one cannot run extraction on, so we have to remove them explicitly from the result set.

3.3 User privileges

This method of pulling information from the system views requires a very small set of rights. To be able to connect to a database, one must have `LIST` privilege on it. It can be given like this:

```
GRANT LIST ON STORE TO flowuser;
```

Now the user `flowuser` can see the `STORE` database in the `_v_database` system view and connect to it. Together with this permission our user also automatically gains `LIST` right on the default schema of `STORE`, otherwise he/she would not be able to connect to that database at all.

It is not strictly necessary to have a permission to connect to an other-than-default schema to see objects from it in the system views. However, it is better to have it, because some names will look shorter and better. For instance, if we are in the DB and schema with a view we want to extract, and this view was created from a table from the same DB and schema, its definition (the inner part of the `SELECT` query found in `_v_view`) will contain one level column names (like `CUSTOMERS.ID`, `CUSTOMERS.NAME`) instead of two level ones (like `SALES.CUSTOMERS.ID`, `SALES.CUSTOMERS.NAME`) – if we are in the same place, it is not necessary for Netezza to explicitly specify the name of the schema.

To see objects in system views, we need `LIST` permission for every one of them (that is, for every table, view and any other user-defined item). We do not require anything else for them, not even `SELECT` privilege, so the data inside will not be viewed or modified.

Naturally, to be able to get metadata for an object which we have `LIST` privilege for, we need `SELECT` permission on every system view listed in Section 3.2.3. Obtaining those rights should not be a problem because every new user gains them by default to have the ability to learn what objects are in the database.

3.4 Requirements

Here we describe the set of requirements our extractor has to fulfill.

3.4.1 Functional requirements (FRs)

FR1: Connecting to and switching between databases
The extractor is supposed to connect to a Netezza host and extract metadata from databases available on it. A potential problem here is to find a way to switch between different databases in runtime. Luckily, in Netezza, it is possible to do it with the `SET CATALOG` command. All we need is an "entry point" – a DB to start with.

FR2: Choosing databases and schemas to extract
The extractor must have a setting to choose what locations it should extract (from all those it sees among accessible DBs and schemas). As we will see in the next chapter, it will be solved with a helper class called `DatabaseSchemaFilter`, which is included in Manta Flow.

FR3: Generation of simplified create scripts
The first of the two main features this project should have is the generation of DDL scripts. Based on gathered metadata, it must build scripts in a simplified yet grammatically correct manner. Later they will be analyzed and processed by other modules of Manta Flow.

FR4: Writing create scripts on disk
The user should be able to choose where the generated create scripts will be saved. A proper folder structure has to be maintained: on the root level, there should be folders representing databases. In them (on the second level) directories denoting extracted schemas must be placed. Finally, on the third level, there should be individual scripts; every one of them must contain info necessary to build one object (a table, for instance).

Also, every file must be given a proper name valid in both Windows and Linux. In the former OS names are more restrictive than those in the latter one, so we will focus on providing compatibility with Windows.

In Netezza, tables with identifiers `MYTABLE` and `MyTable` are distinct objects, but Windows does not see a difference in file names that vary only in their case. Another limitation is that file names in this OS allow fewer types of characters than identifiers of Netezza objects. To sum up, every Netezza name should be unambiguously mapped to a valid Windows one.

FR5: Filling Manta Flow dictionary
The second of the two main features of this extractor should be the ability to fill a special Manta Flow internal structure – a dictionary. The dictionary keeps records of available objects. It is slightly more complicated to properly put items in this data structure than to generate regular create scripts. For example, to save a synonym to it, we must firstly ensure that the target object is already present in it; then we retrieve it and save a pointer to it in the dictionary record of the synonym.

FR6: Choosing objects to extract by type
The user should be able to choose which objects are supposed to be extracted and saved to the dictionary by their type. In the case of procedures returning a result set and synonyms, their reference tables and target objects respectively should be extracted too (no matter if their types are set for extraction or not).

FR7: Choosing objects to save on disk by type
The user should be able to choose which objects are supposed to be saved as create scripts on disk by their type. If some particular type specified here is not among the types which are set for extraction, then such objects will be ignored. In other words, the set of object types to save DDL scripts for must be a subset of the set of object types to extract.

3.4.2 Non-functional requirements (NFRs)

NFR1: Integration with Manta Flow
The extractor is supposed to be a part of Manta Flow tool – it should be able to be seamlessly integrated into it. This can be achieved by creating a class that will implement `InputReader` generic interface passing it a `NetezzaDataDictionary` type – they are both predefined in this software.

NFR2: Lower memory consumption (sequential approach)
Because production databases can have a huge amount of defined objects, it will pay off to work with them sequentially. If we firstly read all of them and only then start the processing, our program will consume an unnecessarily big amount of memory. That is why we will rather perform the extraction step by step: for instance, we can select all distinct names of all stored procedures, then for every name we can get all lists of arguments (let's not forget executables in Netezza can be overloaded), then for every signature formed of the name and the arguments we can finally get the body and the return type. Of course, with such an approach we will need many more SQL requests (and

more time for all of them all to complete), but it is still better to have it this way, so our extractor can run even on lower-end computers.

3.5 Use case

Use cases represent possible interactions that involve our software; they are sets of actions and scenarios, which are linked to certain goals that we can achieve by using our program.

Launch extractor
There is only one main self-explaining use case: launch extractor. However, it supports setting some additional parameters:

- Set dictionary: the user can give our extractor a dictionary, to which database objects will be saved. If it is not done explicitly, a new empty in-memory dictionary will be created automatically before the extraction.
- Set output folder: the user must choose where the generated create scripts will be placed. It may seem that this setting is absolutely necessary and it should be marked with an *include* label rather than with an *extend* one in the diagram; but if the user sets no object types for generation of DDL scripts, this parameter will not be checked, so strictly speaking it is not compulsory.
- Choose locations to extract: see functional requirement #2.
- Set objects to extract: see functional requirement #6.
- Set objects to create DDL scripts for: see functional requirement #7.

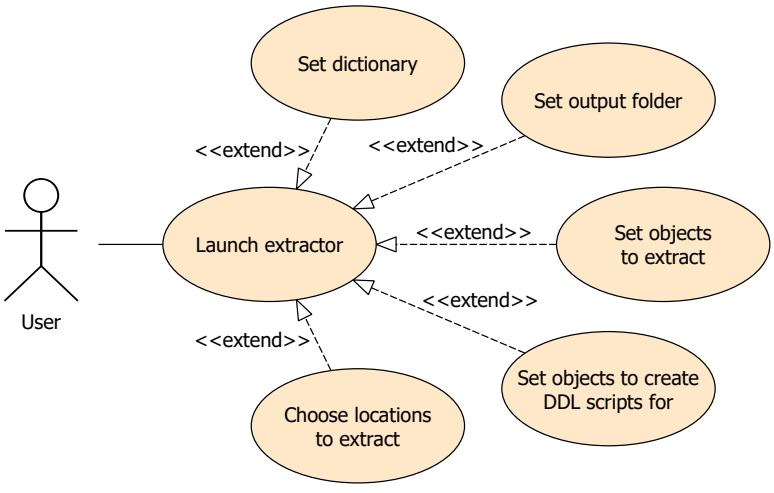

Figure 3.1: Use case diagram

Chapter 4

Realization

In this chapter we will talk about the implementation of our extractor. Firstly, in Section 4.1, we will list technologies and libraries we have chosen. We will also describe their usage in this project. Then, in Section 4.2, we will outline the main parts of the application and how they work together.

4.1 Choice of technologies and frameworks

4.1.1 Programming language

As the main programming language we chose Java. The major parts of Manta Flow are implemented with the help of it, so it will be easy to integrate a module written in it into this software. Another reason is that the author has certain experience in Java programming and sees no major disadvantages of this choice for this work.

4.1.2 Libraries

MyBatis
As stated in section 3.2, we will extract metadata from system views. For this, we need a convenient library to access the database. Our choice here is MyBatis. It is not an ORM framework – instead, it is oriented towards the SQL-first approach. It requires us to write SQL queries and then just maps their results onto objects. Considering the fact that our `SELECT`s are relatively simple, using plain SQL is going to be intuitive and will save us time setting up and configuring a fully-fledged (and sometimes overcomplicated) ORM solution.

MyBatis uses so-called *mappers* – special objects designed to run queries in a database. A mapper is defined as Java interface, every method of which has an SQL query associated with it. Queries can be written either in-place

(in annotations above method definitions) or in a separate XML file. We will use the second approach because it is more flexible and less messy.

Maven
In order to be easily included as a dependency into the bigger piece of software and to manage our own dependencies (such as, for instance, MyBatis or `NetezzaDataDictionary`, which is defined in another artifact of Manta Flow) we require a good dependency manager. We chose Maven – it is a tool for build automation. It enables us to not only configure dependencies of our project but also to get control over how the extractor will be built and deployed. Moreover, we can set different properties, such as the name of the artifact, its version, etc.

JUnit
JUnit is a framework to write tests for Java programs. It provides a simple mechanism for creating and running tests on different parts of software, which will be run (by Maven) every time we build the project. Firstly, we can use them to demonstrate that our extractor is working properly. Secondly, when its source code is modified, and these tests are run again, we will see if the extractor still functions correctly (all tests are successful) or something was broken (one or more tests fail). We will create separate tests both for the application as a whole and its individual parts.

Spring
Spring Framework (or simply Spring) is a framework for Java applications. It is modular and has many different packages with different functionality that make building more flexible and functional software easier. We will use several:

- *spring-core* and *spring-context* are among the main parts of Spring. The most important thing we will use from them is the inversion of control (IoC) container. We will have one or more XML configuration files that will help us put different parts of the extractor together (another way to do it is to use annotations or special configuration classes in Java, but because Manta Flow uses XML it will be easier to integrate the extractor in it if we choose this strategy).

  That way we will take advantage of loose coupling – various functional parts of our application will know as little as possible about each other, so it will be simpler to manage, test and, if needed, correct or even replace them. "Loose coupling ... enables us to write more maintainable code" [18].

- *spring-mybatis* will integrate MyBatis with Spring (since Spring will be the main framework of our extractor). It will simplify the creation of mappers, let MyBatis use Spring transactions, and much more [19].

- *spring-test* will make the context of our application available in JUnit tests – in several cases it is required to correctly wire beans we want to test. A bean is one of the main objects in the program that is taken care of by Spring IoC container [20].

Netezza JDBC
To access the database from a Java application, we can use a Java Database Connectivity (JDBC) implementation designed to work with Netezza – Netezza JDBC driver. It can be downloaded from IBM website (together with Netezza documentation and some other utilities). It will allow MyBatis to run SQL statements and get data in response.

4.2 Project structure

4.2.1 Classes

Here is the class diagram that shows the main parts of our extractor.

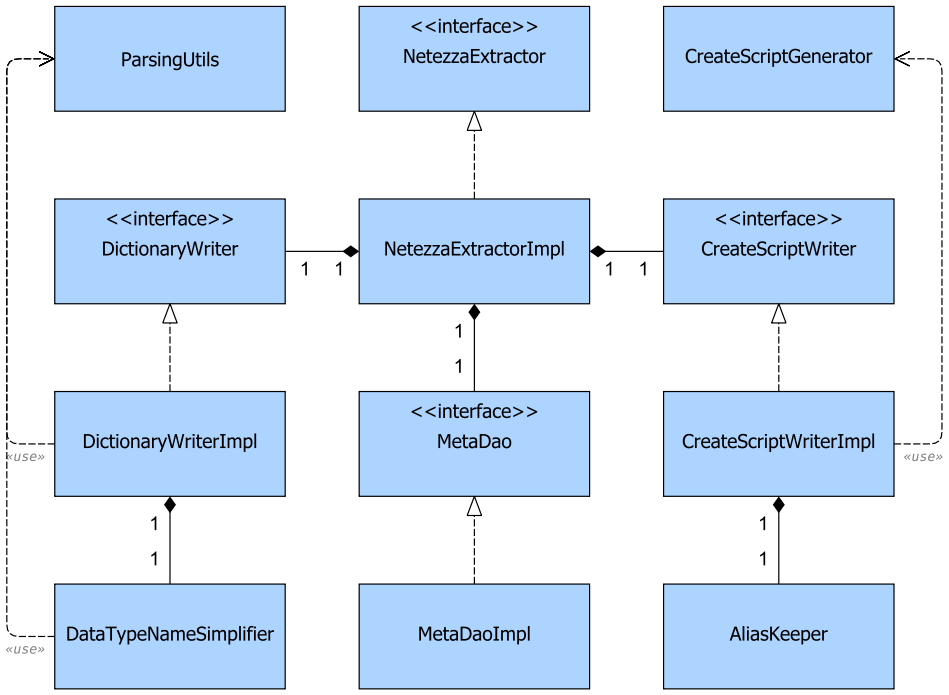

Figure 4.1: Class diagram

`NetezzaExtractorImpl`
This is the main class of our application. It contains the major part of the logic of the program. It controls and makes use of nearly all other classes.

Apart from different setters that help configuring its flow, it has a public method `extract()` that launches the extractor. Firstly, it checks if the set called `outputDdlTypes` is empty. If it is not (the user wants to generate create scripts for objects of at least one type, see FR7), it checks that a `File` object denoting the root folder was given before the start of `extract()`. If it was, then this folder is ensured to be empty (created if it does not exist or cleared if it had some files already). If the folder is `null` or if an error occurs during creating or clearing it (for the lack of permissions or if there is a regular file occupying its name instead), an error is logged and the process is stopped.

Then it retrieves all databases and all their schemas it can see. The system ones (discussed in Section 3.2.3) are removed explicitly. So are removed the locations that are not set for extraction by the user – this is done with the help of `includeFilter` and `excludeFilter` of `DatabaseSchemaFilter` type. This helper class is defined within Manta Flow and enables us to choose what databases and schemas on the host server we want to extract. A list of only relevant locations is logged into the info channel.

Now, for every database and schema it extracts metadata of all objects of types from the set named `extractedDdlTypes` (see FR6). Data structures of different type can be found in different system views, and if they happen to be in the same one (like tables and external tables), they can be additionally filtered with an appropriate `WHERE` clause (as described in Section 3.2.3). It means that no redundant `SELECT` statements are executed. The process goes sequentially (see NFR2).

Every retrieved object is checked against `outputDdlTypes` set and, if the user wants to save it, the extractor generates a DDL script for it and places it on disk. Then it is also put to the dictionary – this process is not supposed to be always successful on the first try. The thing is, there are two kinds of dependencies in Netezza:

- Stored procedures that return a result set: as a return type they have a fully qualified table name. The result set by its structure will be akin this table, so the latter must be already saved in the dictionary. We must get it from it and copy inside the dictionary record of the procedure. If the table is in the DB or schema which are not supposed to be extracted, a dummy table type will be created and saved instead. If it is in a "valid" location that was simply not yet processed, then a dependency will be generated and left until the end of the extraction.
- Synonyms: similar to the case above, saving a synonym requires its target to be present. If it is not yet in the dictionary, but its location is "valid", a normal dependency is generated and left for later. But if we are not going to extract it, we do not want to save such a synonym at all (and all other objects that are dependent on this one). That is why a special kind of relation is generated – a dead dependency (the term is probably arguable, but this is how it is called in this project implying that it will

never be satisfied). It is used to block the extraction of objects that can possibly depend on this synonym.

Dependencies are kept in the class `DependencyManager` in a graph-like structure.

It is worth noting that, according to the documentation, there can be only one-level dependencies. But in practice, for instance, we can create a chain of synonyms up to the depth of 16. For that reason we cannot just skip writing a synonym with an unknown target to the dictionary – we must block every other synonyms dependent on it. However, it does not prevent us generating and saving DDL scripts.

In the very end of the extraction process, objects (or, more precisely, the info about their locations) that have dependencies will be one by one removed from our internal graph and selected from the DB explicitly.

If during the process a one-level dependency is added and later, but before the end of the extraction, while pulling out metadata from some other place, it is successfully retrieved, it will not be queried again: after the extraction of every object, we check, if it is not an "opened node" of our graph (an independent object by itself). If it is, we simply mark it as extracted by removing it from the graph.

If there were any unsatisfied dependencies left, like those dead ones or cycles (with Netezza's synonyms it is indeed possible), they will be listed through the warn channel of the logger.

`MetaDaoImpl`

`MetaDao` stands for metadata access object. Its primary purpose is to query databases and get info from them. As private members, it has MyBatis mappers – Java interfaces with SQL statements (written in separate XML files) corresponding with their methods. Every mapper is designed to handle object of one particular type, that is why they are called `TableMapper`, `ViewMapper`, etc.

Most methods of `MetaDaoImpl` rather straightforwardly conform to those of different mappers. However, there are a few places that implement slightly more complicated logic:

- `getAllDatabasesWithSchemas()`: this method not only pulls out every visible database with its schemas but also removes items that we do not want to extract from the list.

  There is one special mapper, which is used only in the constructor of `MetaDaoImpl`, – `HelperMapper`. By executing its only `SELECT` statement, it returns the current default system case (upper or lower one). With the help of it, we can correctly identify and remove unwanted system items (i.e. if we have uppercase, we will remove the database called exactly `SYSTEM`, not `system` or `System`, because in an uppercased Netezza the former is a system database and the latter two can be user-defined ones).

After removing system items, the method checks every database and schema against filters and removes those the user has marked as not to be processed. Finally, in the returning list, there are only "valid" locations that will later be extracted.

- `determineObjectsTypes()`: this auxiliary private method is called when we select a synonym. Because from Netezza's synonym metadata it is impossible to deduce what type is the target object of, we perform `SELECT` requests on all system views in its location. We also need to keep in mind that a synonym can point to an overloaded executable, so if the target is a procedure or a UDX, we also select and save all argument lists of it to be able to unambiguously identify the target (to get it from a dictionary or to firstly retrieve it later without scanning for it).

  If the target of the synonym is non-existent or located in a DB or schema we are not supposed to extract, we set its target type as `UNDEFINED`, so later it will not be saved to the dictionary and will block other synonyms that could possibly be dependent on it.

When synonyms are already retrieved but not yet given to the method discovering their target type(s), they are sorted by the database they point into. We made it to reduce the number of switching between different DBs.

`CreateScriptWriterImpl`
This class is used to saving DDL scripts to files on disk. When writing one, methods of this class make sure that all necessary directories exist. The path is given as a list of strings: the last one of them is treated as the name of the future file with the script, all others preceding it – as names of folders relative to the base folder (the root which the user must provide before running the extractor). This class also converts all Netezza identifiers to their Windows-friendly analogs with the help of an instance of `AliasKeeper`.

`AliasKeeper`
`AliasKeeper` is designed to generate Windows-friendly names from Netezza identifiers and keep track of them. For every new identifier it creates a valid filename and, if it is already occupied, adds a suffix like `_2`, `_3` and so on until a free option is found. For example, if we have tables named `My*Table` and `My?Table`, both of them will be transformed into the `My_Table` (characters that are illegal in Windows are replaced with underscores) and then the second one will be given a suffix turning it into `My_Table_2`.

The position of the suffix depends on whether we have a file name that also ends with `.sql` or something else. If it is the first case, we add it right before the `.sql` extension (like `table_2.sql`). Otherwise, we put it simply at the end of the name (a folder like `schema_2`).

This mechanism enables us to map every required Netezza identifier to a Windows-friendly one unambiguously.

After every extraction, all aliases are cleared so the database can be changed in any way before the next launch of the extractor.

`CreateScriptGenerator`
This class has only static methods. They take objects that internally represent real objects extracted from Netezza and generate appropriate DDL scripts. These scripts are simplified and cannot be used to completely restore the database because they can possibly lack some information (like column constraints, for example, – for the current version of Manta Flow we do not extract any constraints at all), but they conform to the Netezza syntax and can be processed with other modules for their code to be statically analyzed.

A note related to UDXs – create scripts for them are not generated. They are written in C++, and it is impossible to retrieve their source code, so there is nothing to work with later. Still, UDXs are saved to the dictionary.

`DictionaryWriterImpl`
The main purpose of `DictionaryWriterImpl` is to write extracted metadata to the special structure of Manta Flow called a dictionary. When parsing DDL scripts, it is used to look up objects defined in the database to see if they exist and what properties they have.

The way that this class operates with the dictionary is given by the built-in API of the latter. It was designed to be database-independent and to work with various solutions; however, when it came to Netezza, it appeared to have certain limitations. Unfortunately, they cannot be quickly resolved without significant interventions in the structure of the dictionary module, which will probably break other existing parts of the software that use it.

One such limitation is the impossibility of saving two overloaded executables with the same number of arguments (arguments themselves are of course different, otherwise these executables could not be considered overloaded). So when saving a procedure or a function, we always check if something with the same name and number of arguments is already in the dictionary. If it is, we do not save it again; instead, we log the info about this event to the warn channel and immediately leave the method.

Another limitation is that we cannot set multiple targets to one synonym. In that case, we create as many synonyms as there are targets (minus those overloaded executables that themselves cannot be placed in the dictionary).

Despite the drawbacks described above, it is absolutely possible to work with the dictionary in a correct way and fill it with the majority of objects that are present in the database. Items that will not be written to it will be at least logged. In future, this situation will likely to change for the better.

`ParsingUtils`
This class has a few static methods that come in handy when we need to parse something. For example, we can get a list of `Column` objects from their raw definitions (in case we need to create a table type as a return value while saving

a UDTF to the dictionary). Or we can generate a list of data types without parameters from a single string (for instance, if an executable has a string `"(NUMERIC(5,2), VARCHAR(50))"` representing the list of its arguments, we will obtain an array of strings with two items – `"NUMERIC"` and `"VARCHAR"`).

ParsingUtils is mainly used by the `DictionaryWriterImpl` class. Also, its ability to remove parameters from a single data type definition is utilized in `DataTypeNameSimplifier` class, described below.

`DataTypeNameSimplifier`
Our extractor should work well with other parts of Manta Flow, so it needs to conform to some special requirements. One such requirement is to convert all data types to the six main ones currently supported by resolver (see Appendix B). For example, all integer types must be converted to `NUMERIC` (before saving them to the dictionary), all string types (such as `NCHAR`, `TEXT`, `VARCHAR`) – to `CHARACTER` and so on. Here our simplifier comes to help.

Actually, there is only a single map encapsulated in this class (and a method, which accepts a data type, then with the help of `ParsingUtils` removes parameters from it, and returns a value that represents a simplified type name from the map). The reason to create a separate class for this functionality is the long initialization of the map: every data type as a key and every its simplified analog as a value must be put inside it.

4.2.2 Predefined structures

In our project, we have three classes which are not written by the author of this thesis. They assist in the integration of the extractor into Manta Flow.

`NetezzaExtractorReader`
As we discussed earlier (see NFR1), in order to integrate the extractor into Manta Flow, we must create a class implementing `InputReader` interface. So there is one more class, not present in the diagram, that does exactly this. It has the `NetezzaExtractor` as a private member (given through a setter), and its `read()` method, which should be implemented by us, simply checks if there is a dictionary (and generates a new in-memory empty one, if there is not), launches the extractor by calling its main `extract()` method, and finally returns the dictionary filled with metadata.

`DdlType`
This is simply an enum with all possible object types that we want to distinguish in our extractor. It also has a special `UNDEFINED` value, which is used when we cannot find out the type of the target object of a synonym.

`NetezzaExtractor`
This is the main interface we need to implement. It defines all necessary methods our extractor should have. It is realized by `NetezzaExtractorImpl` class.

4.2.3 Configuration

The main framework of the application is Spring. As in other modules of Manta Flow, the configuration of the extractor is written in XML. It is divided into two main files:

- The first one is located in the `main` package. It has bean definitions of:
  - `MetaDaoImpl`
  - `CreateScriptWriterImpl`
  - `DictionaryWriterImpl`
  - All the mappers
  - `SqlSessionFactoryBean`
  - `DataSourceTransactionManager`

  The last two classes are additional helper elements necessary for our module to be able to access the database.
- The second one is in the `test` package. It contains bean definitions of:
  - `NetezzaExtractorImpl`
  - `BasicDataSource`

  The last class, for example, assists in establishing a connection to the database by sending the details about it (such as user credentials) to the DB server.

This way, when used as a standalone application, the extractor has all needed configurations to run, they are just in two places. This is useful for development and testing purposes. When it is connected to Manta Flow, the second configuration file is not used; instead, the data source and the main class are set up differently, according to the production environment. Beans from the first XML do not require any special attention, so there is no point in changing them. It allows us to import the first file as is and then add only two bean definitions ourselves.

Chapter 5

Testing

In this chapter, we will describe the tests our application has.

To create automated tests, we use JUnit library (as described in Section 4.1.2). They are run by Maven automatically on every build. This way we can have some reassurance that if, after any changes to the code, they still complete successfully, the whole application will most likely work properly.

5.1 Test scenarios

Here is a list of tests we have:

`AliasKeeperTest`
The main purpose of `AliasKeeper` is to generate Windows-friendly file names for all Netezza identifiers unambiguously. That is why in this test we supply it with several potentially troublesome object names that differ from each other, but not too much. Moreover, if to convert them straight to their Windows analogs, they will collide. So we give them to `AliasKeeper`, take the results it produces and put them in a set (which will not accept duplicates). We compare the size of this set to the number of the original names. If they are equal, then our `AliasKeeper` works correctly as (with the help of suffixes) it does not generate colliding values.

`CreateScriptGeneratorTest`
In this set of tests we check the rightness of generated create scripts. We give different objects to our generator, take its output and compare it with the hardcoded values, which are considered the right ones. In order to eliminate problems with different indentation and number of whitespace characters in the scripts, we firstly normalize both of them by replacing every sequence of any white spaces in them with only one space. Then we compare the output of the generator with the hardcoded alternative, and if they are equal, everything is in order.

`CreateScriptWriterTest`
Here we test the correctness of our writer of DDL scripts. We use a temporary folder from JUnit not to litter the file system and not to think of real paths and permissions for them (also applies in an environment of a continuous integration system). We ask our writer to ensure there is an empty folder, which will later be filled with scripts. Then we check if it exists and is empty. We fill it with some garbage files, then ask the writer once again to provide a free place for us. It should delete the contents of the given folder. Finally, we ask it to create a file for us with a script. We check that the file was created, its path is correct, and its contents are the same that we provided.

`DependencyManagerTest`
To test how our program manages dependencies we create several possible graphs of dependent objects by adding edges one by one. Then we remove objects (also stepwise) and check the order in which our `DependencyManager` returns them. If it is correct, then it works as expected.

`DictionaryWriterTest`
In this set of tests, we check the correctness of how the extracted objects are written into the dictionary of Manta Flow.

We use a method with `@BeforeClass` annotation to statically initialize the dictionary for all future test cases, and after each one, we call `forgetChanges()` to clear it. This is done because creating a new instance of this structure is a relatively expensive procedure, and there is no point in generating a new one for every test.

We save objects of various types to the dictionary and then retrieve them from it to check their attributes. We look at their names, types, and some other properties. If they are equal to those from original objects, then the saving mechanism works properly.

`MetaDaoTest`
Here we test how metadata is selected from the database. A running copy of Netezza is required to perform those cases. We retrieve different structures from our test database and then compare the objects they are mapped to by MyBatis with the hardcoded ones – they should be equal.

`ExtractorTest`
This test targets the whole application – it checks the job done by the realization of `NetezzaExtractor` interface. After the extraction is complete, we compare both generated create scripts and objects saved to the dictionary with their hardcoded versions. Similarly to `MetaDaoTest`, a running copy of Netezza is necessary.

As you can see, we have several tests targeting individual components of our application and one big test for the extractor on the whole. It enables us to test our software much more granularly, so we have more confidence that

everything is working how it should, and if it is not, we can find and fix bugs earlier.

5.2 Test data

We have two files with data necessary for our test scenarios:

`TestResources`
Here we have all hardcoded data. We have objects of different database structures and DDL scripts for them, which are considered to be correct. During tests, we compare real outputs of the program to them. This class extends `ExternalResource` from JUnit library and is instantiated only once for every test file using it with the help of `@ClassRule` annotation.

`TestDatabaseDdlScript`
This is a script that can be used to recreate our test database on which tests of `NetezzaExtractorImpl` and `MetaDaoImpl` are run. Although it is easier to simply disable these scenarios than to install Netezza and keep it running during every build of the extractor, we highly recommend not doing this. The script contains all required DDL statements and can prepare the database with all objects in one run.

Chapter 6

User documentation

Here is a user's manual, which will describe the actions one needs to perform to use our project.

The steps are as follows:

1. Create a new Java project, which will use the extractor as a module.
2. Add the extractor as a dependency (either through IDE or with the help of a build system, such as Gradle or Maven).
3. Add spring-core and spring-context as dependencies (our application uses XML Spring configuration to wire up its parts together, but you will need to extend it by defining two more beans yourself).
4. Import `NetezzaExtractorBase.xml` file with the main parts of the configuration.
5. Define `DataSource` as a bean with ID "dataSource". The extractor was tested with `BasicDataSource` from Apache's DBCP, but other implementations can potentially work too.
6. Define `NetezzaExtractorImpl` as a bean. The beans with following IDs should be provided as arguments to the constructor.
   - `metaDao`
   - `scriptWriter`
   - `dictionaryWriter`
7. Use setters to give `NetezzaExtractorImpl` additional parameters, such as an empty dictionary (required), root folder for DDL scripts, etc.
8. Now you can launch the module by running its `extract()` method.

On the attached CD you can find a folder `extractor-test-project` – this is a little program that shows a concrete example of how to add our extractor to another project.

Both pieces of software can be compiled (firstly the extractor, then the demo) using the following Maven command:

```
mvn clean install
```

Please note, that there is not so much use of our application this way (without Manta Flow). For example, the dictionary will be filled for nothing unless your program knows how to use it. Generated create scripts are designed to conform to the requirements of Manta Flow, so, to you, they can appear slightly incomplete.

Note also that the application on the CD attached to this thesis cannot be compiled without several dependencies, which were developed by Manta team and legally belong to it (the dictionary, for example). We cannot distribute them freely, so these modules are not on the CD. The source code of the extractor itself, however, is fully present.

Conclusion

The aim of this thesis was to gain knowledge about the kinds of metadata in Netezza and ways to retrieve them in order to design, implement and test a new Netezza-specific extractor module for Manta Flow. The author considers these targets fully achieved.

In the first part, we explored the environment of the given RDBMS: we got familiar with the objects that can be found there, the data that can be used to describe them and the possibilities of extracting it. Then, we designed the requirements for our application, chose technologies for it and used this information to put it into life. Finally, we created automated tests for it and a user manual, which can help including the extractor into a third-party project and launching it.

Although this work does not complexly solve the problem of analysis and visualization of the structure of Netezza databases, it lays the foundations for it: other modules can use the outputs of the extractor as a source of data to work with. And even if they had been developed in advance, the whole solution would still be incomplete without our project. The results of the theoretical part can help to better understand the objects that are present in Netezza and ways to get their descriptive info while designing other software.

There are several ways to enhance our application in future. Firstly, the set of extracted metadata can be extended. For now, there is no such necessity, but later, if Manta Flow needs (or can process) something else, parts of currently ignored information can be put to use. Secondly, our module can be redesigned to work with another complex software solution. It will most likely mean big changes to the code, but it is possible, especially if a good and simple API is provided.

Bibliography

[1] Reference. *Why do companies use database? [online].* IAC Publishing, LLC, ©2017 [viewed 17 February 2017]. Available from: `https://www.reference.com/technology/companies-use-database-e0c5d2ba994c2360`

[2] Kreibich, J. A. *Using SQLite*. Sebastopol: O'Reilly Media, Inc., August 2010, ISBN 9780596521189.

[3] WhatIs.com. *Metadata [online].* TechTarget, ©1999–2017 [viewed 19 February 2017]. Available from: `http://whatis.techtarget.com/definition/metadata`

[4] Sampagar, V. *Netezza TwinFin Architecture [online].* 2 July 2016, DWgeek.com, ©2016 [viewed 24 February 2017]. Available from: `http://dwgeek.com/netezza-twinfin-architecture.html/`

[5] IBM [online]. *IBM Netezza Database User's Guide*. Revised: 16 September 2014. IBM Corporation, ©2011, 2014 [viewed 28 February 2017]. Available from: `https://www-01.ibm.com/marketing/iwm/iwm/web/reg/download.do?source=swg-im-ibmndn&lang=en_US&S_PKG=d12&cp=UTF-8&&&dlmethod=http`

[6] IBM [online]. *IBM Netezza Data Loading Guide*. Revised: 15 September 2014. IBM Corporation, ©2011, 2014 [viewed 28 February 2017]. Available from: `https://www-01.ibm.com/marketing/iwm/iwm/web/reg/download.do?source=swg-im-ibmndn&lang=en_US&S_PKG=d12&cp=UTF-8&&&dlmethod=http`

[7] IBM [online]. *IBM Netezza Stored Procedures Developer's Guide*. Revised: 16 September 2014. IBM Corporation, ©2009, 2014 [viewed 28 February 2017]. Available from: `https://www-01.ibm.com/marketing/iwm/iwm/web/reg/download.do?source=swg-im-ibmndn&lang=en_US&S_PKG=d12&cp=UTF-8&&&dlmethod=http`

[8] IBM [online]. *IBM Netezza User-Defined Functions Developer's Guide*. Revised: 16 September 2014. IBM Corporation, ©2007, 2014 [viewed 28 February 2017]. Available from: https://www-01.ibm.com/marketing/iwm/iwm/web/reg/download.do?source=swg-im-ibmndn&lang=en_US&S_PKG=d12&cp=UTF-8&&&dlmethod=http

[9] W3Schools. *SQL Constraints [online]*. Refsnes Data, ©1999–2017 [viewed 5 March 2017]. Available from: https://www.w3schools.com/SQl/sql_constraints.asp

[10] Birmingham, D. Distribution – what's up with that? In: *IBM developerWorks [online]*. 6 July 2011, IBM Corporation, ©2012 [viewed 6 March 2017]. Available from: https://www.ibm.com/developerworks/community/blogs/Netezza/entry/distribution_what_s_up_with_that13?lang=en

[11] Coffing, T.; Nolan, J. *The Brilliance of Netezza*. Coffing Publishing, April 2014, ISBN 9781940540276.

[12] Murray, C. *Oracle Database 2 Day Developer's Guide [online]*. January 2017, Oracle and/or its affiliates, ©1996, 2017 [viewed 12 March 2017]. Available from: https://docs.oracle.com/database/122/TDDDG/toc.htm

[13] h_wiedey. Comparison of Triggers in MS SQL and Oracle. In: *Code Project [online]*. 29 July 2013, [viewed 19 March 2017]. Available from: https://www.codeproject.com/articles/621532/comparison-of-triggers-in-ms-sql-and-oracle

[14] LeBlanc, P. *Microsoft SQL Server 2012 Step by Step*. Microsoft Press, 25 February 2013, ISBN 9780735663862.

[15] *SQL Server Database Engine [online]*. Microsoft, ©2017 [viewed 20 March 2017]. Available from: https://technet.microsoft.com/en-us/library/ms187875(v=sql.110).aspx

[16] Tolpeko, D. *CREATE TYPE – User-Defined Types – SQL Server to Oracle Migration [online]*. October 2013, SQLines, ©2010–2017 [viewed 26 March 2017]. Available from: http://www.sqlines.com/sql-server-to-oracle/create_type

[17] IBM [online]. *IBM Netezza System Administrator's Guide*. Revised: 15 September 2014. IBM Corporation, ©2001, 2014 [viewed 28 February 2017]. Available from: https://www-01.ibm.com/marketing/iwm/iwm/web/reg/download.do?source=swg-im-ibmndn&lang=en_US&S_PKG=d12&cp=UTF-8&&&dlmethod=http

[18] Seemann, M. Dependency Injection is Loose Coupling. In: *Ploeh blog [online]*. 7 April 2010, Mark Seemann, ©2017 [viewed 2 April 2017]. Available from: `http://blog.ploeh.dk/2010/04/07/DependencyInjectionisLooseCoupling/`

[19] *MyBatis-Spring: Introduction [online]*. MyBatis.org, ©2010–2017 [viewed 8 April 2017]. Available from: `http://www.mybatis.org/spring/`

[20] TutorialsPoint. *Spring – Bean Definition [online]*. ©2017 [viewed 9 April 2017]. Available from: `https://www.tutorialspoint.com/spring/spring_bean_definition.htm`

Appendix A

Acronyms

API Application programming interface

CD Compact disc

CPU Central processing unit

DB Database

DDL Data definition language

DML Data manipulation language

FPGA Field-programmable gate array

FR Functional requirement

ID Identifier

IDE Integrated development environment

IoC Inversion of control

MPP Massively parallel processing

NFR Non-functional requirement

NPS Netezza Platform Software

NZPLSQL Netezza procedural language (extension for SQL)

OLE DB Object Linking and Embedding, Database

ORDBMS Object-relational database management system

ORM Object-relational mapping

OS Operating system

PL/SQL Procedural Language/Structured Query Language

RAM Random-access memory

RDBMS Relational database management system

SQL Structured Query Language

T-SQL Transact-SQL

XML Extensible markup language

Appendix B

Data type mappings

Here is a complete list of mappings used to reduce the amount of data types in the dictionary:

Table B.1: Data type mappings

Source type	Resulting type
BYTEINT | DEC INT1 | INT2VECTOR SMALLINT | REGPROC INT2 | OID INTEGER | TID INT | XID INT4 | CID BIGINT | OIDVECTOR INT8 | SMGR DECIMAL | _INT4	NUMERIC
FLOAT | DOUBLE FLOAT4 | DOUBLE PRECISION REAL | FLOAT8	FLOAT
CHAR | VARCHAR CHARACTER | UNKNOWN BPCHAR | CHAR VARYING TEXT | CHARACTER VARYING	CHARACTER

NATIONAL CHARACTER NATIONAL CHAR NCHAR NATIONAL CHARACTER VARYING NATIONAL CHAR VARYING NVARCHAR	CHARACTER
BOOLEAN | BOOL	BOOLEAN
DATE | DATETIME TIME | TIMESTAMP TIMETZ | TIMESPAN DATE | DATETIME TIME WITH TIME ZONE TIME WITHOUT TIME ZONE	DATETIME
VARBINARY | BINARY ST_GEOMETRY | BYTEA UNKBINARY | BINARY VARYING	BINARY

Appendix C

Contents of enclosed CD

- `readme.txt` ..... the file with CD contents description
- `thesis` ..... the folder with the thesis
  - `BP_Laskov_Boris_2017.pdf` ..... the thesis in PDF format
  - `src` ..... source codes of the thesis in LaTeX
- `apps` ..... the folder with Java projects
  - `netezza-dictionary-extractor` ..... sources of the extractor
  - `extractor-test-project` ..... sources of the test project